

# Volume 4B

# **Program Development Tools**

---

---

---

## **Volume 4B. Program Development Tools**

**#996042**

Copyright © 1984, Symbolics, Inc. of Cambridge, Massachusetts. All rights reserved.  
Printed in USA. This document may not be reproduced in whole or in part without the  
prior written consent of Symbolics, Inc.

Design: Schafer/LaCasse

Cover and title page typography: Litho Composition Co.

Text typography: Century Schoolbook and Helvetica produced on a Symbolics 3600

Lisp Machine from Bitstream, Inc., outlines; text master printed on Symbolics

LGP-1 Laser Graphics Printer.

The first Lisp Machine system was a product of the efforts of many  
people at the M.I.T. Artificial Intelligence Laboratory, and of the unique  
environment there. Portions of earliest versions of many of the documents  
in this documentation set were written at the AI Lab.

# Contents

---

## **Program Development Tools**

---

### **ZMACS**

Zmacs Manual

### **DEBUG**

Debugger

### **MAINT**

Maintaining  
Large Systems

### **COMP**

The Compiler

### **MISCT**

Other Tools

*symbolics*™

---

# ZMACS Zmacs Manual

---

---

Cambridge, Massachusetts

---

# Zmacs Manual

# 990050

February 1984

**This document corresponds to Release 5.0.**

This document was prepared by the Documentation Group of Symbolics, Inc.

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Symbolics is a trademark of Symbolics, Inc., Cambridge, Massachusetts.

UNIX is a trademark of Bell Laboratories, Inc.

PDP-10 is a trademark of Digital Equipment Corporation

VAX is a trademark of Digital Equipment Corporation

TENEX is a registered trademark of Bolt Beranek and Newman Inc.

Copyright © 1984, 1983, 1982, 1981, Symbolics, Inc. of Cambridge, Massachusetts.  
All rights reserved. Printed in USA.

This document may not be reproduced in whole or in part without the prior written consent of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

## Table of Contents

	Page
<b>Introduction</b>	<b>1</b>
Overview	2
Scope	2
Organization	2
Introduction to Zmacs	4
Overview	4
Commands	4
Keystrokes	4
Extended Commands	5
Command Tables	5
Additional Notation Conventions	7
Documentation Conventions for Commands	7
<b>Getting Started</b>	<b>9</b>
Entering Zmacs	10
Introduction	10
Getting Help in Zmacs	12
Introduction	12
HELP	12
Completion	12
Yanking	13
Organization of the Screen	15
Introduction	15
Editor Window	15
Echo Area	17
Mode Line	17
Inserting Text	20
Introduction	20
Inserting Characters	20
Starting a New Line	20
Correcting Typos	20
Wrapping Lines	20
Inserting Formatting Characters	21
Numeric Arguments	22
Moving the Cursor	23
Description	23
Summary	23
Erasing Text	25
Description	25
Summary	25

<b>Creating and Saving Buffers and Files</b>	<b>27</b>
Description	27
Summary	27
Creating a Buffer	27
Saving a File	28
Creating a File	28
Editing Existing Files	29
<b>Leaving Zmacs</b>	<b>30</b>
Overview	30
SELECT Key	30
System Menu	30
c-z	30
<b>Getting Help</b>	<b>31</b>
Getting Out of Trouble	32
Overview	32
Getting out of Prefixes and Prompts	32
Large Deletions	33
Finding Out About Zmacs Commands	34
Overview	34
HELP	34
Finding Out What a Command Does	34
Searching for Appropriate Commands	35
Finding Out What You Have Typed	36
More HELP Commands	36
General Information-giving Commands	36
The Editor Menu	39
Overview	39
Editor Menu Commands	39
More on the Minibuffer	41
Response Format	41
Response Help	41
More Ways to Enter Responses	41
Mousing	42
Completion	42
<b>Moving the Cursor</b>	<b>45</b>
Overview	46
Introduction	46
The Editor Window	46
Redisplaying the Window	47
Introduction	47
Recentering Window	47
Next Screen	47
Previous Screen	48
Positioning Window Around Definition	48

Moving to Specified Line	48
Using the Mouse	49
Introduction	49
Mouse Documentation Line	49
Motion Commands	51
Introduction	51
Numeric Arguments	51
Motion by Character	51
Motion by Word	52
Motion by Sentence	52
Motion by Lisp Expression	53
Motion by Line	56
Goal Column	57
Motion by Paragraph	58
Introduction	58
Forward Paragraph	58
Backward Paragraph	58
Motion by Page	59
Introduction	59
Forward Page	59
Backward Page	59
Motion with Respect to the Whole Buffer	60
Beginning/End of Buffer	60
<b>Deleting and Transposing Text</b>	<b>61</b>
Deleting vs. Killing	62
Overview	62
Kill History	63
Viewing Kill History	63
Viewing the Editor Command History	63
Using the Mouse on History Elements	64
Retrieving History Elements	64
Kill Merging	64
Deleting and Transposing Characters	66
Deleting the Last Character	66
Deleting the Current Character	66
Transposing Characters	66
Deleting and Transposing Words	68
Introduction	68
Deleting the Current Word	68
Deleting the Previous Word	68
Transposing Words	68
Deleting and Transposing Lisp Expressions	69
Introduction	69
Deleting the Current Expression	69
Deleting the Previous Expression	69



Deleting the List Containing Current Expression	69
Transposing Expressions	70
Deleting and Transposing Lines	71
Introduction	71
Deleting the Current Line	71
Deleting Backward on the Line	71
Transposing Lines	71
Deleting Sentences	72
Introduction	72
Deleting the Current Sentence	72
Deleting the Previous Sentence	72
<b>Working with Regions</b>	<b>73</b>
What is a Region?	74
Introduction	74
Point	74
Mark	74
Creating a Region	75
The Point-pdl	75
Setting/Popping the Mark	76
Moving to Previous Points	76
Showing the Mark	77
Registers	78
Saving and Moving to Locations in Registers	78
Saving and Inserting Regions in Registers	78
Commands to Mark Regions	80
Overview	80
By Words	80
By Lisp Expressions	80
By Paragraphs	80
By Pages	81
By Buffers	81
Region-Manipulating Commands	82
Saving the Region	82
Deleting the Region	82
Compiling the Region	82
Transposing Regions	82
Hardcopying the Region	82
Filling the Region	83
Other Region-related Commands	84
<b>Searching, Replacing, and Sorting</b>	<b>85</b>
Searching	86
Overview	86
Incremental Search	86
Reverse Incremental Search	87

String Search	88
Locating and Replacing Strings Automatically	90
Overview	90
Making Global Replacements	90
Querying While Making Global Replacements	90
Querying While Making Multiple Global Replacements	91
Other Types of Replacements	92
Tags Tables and Search Domains	94
Introduction	94
How They Work	94
Specifying and Listing Tags Tables	94
Performing Operations With Tags Tables	95
Support Buffers	97
Sorting	100
Overview	100
<b>Manipulating Buffers and Files</b>	<b>101</b>
Working With Buffers and Files	102
Overview	102
Buffer and File Names	102
Buffer Flags for Existing Files	103
Buffer Flags for New Files	103
Selecting, Listing, and Examining Buffers	104
Current Buffer	104
Buffer History	104
Buffer Commands	105
Changing Buffers	105
Listing Buffers	106
Editing Buffers	106
Viewing a Buffer	107
Hardcopying the Buffer	108
Renaming the Buffer	108
Writing Out All Buffers	108
Reading a File Into a New Buffer	108
Reading a File Into an Existing Buffer	109
Writing the Buffer Contents to a File	109
Saving the Buffer Contents to the File	109
Re-reading a File Into the Buffer	109
Creating a Fundamental Mode Buffer	110
Associating a File With a Buffer	110
Destroying Buffers	110
Appending, Prepending, and Inserting Text	112
Appending a Region to a Buffer	112
Appending a Region to a File	112
Prepending a Region to a File	112
Inserting a Buffer Into Another Buffer	112

Inserting a File Into a Buffer	112
Comparing Files and Buffers	113
Source Compare	113
Source Compare Merge	113
Compare/Merge Commands for Definitions	115
Window Commands	117
Using Two Windows, Select Bottom	117
Using Two Windows, Select Top	117
Two Windows, Specify Other Contents	117
Two Windows, Region in Top	117
Change Window Size	117
Choose Other Window	118
Return to One Window	118
Scroll Other Window	118
Split Screen	118
File Manipulation Commands	119
Overview	119
Listing Files in a Directory	119
Displaying the Contents of a Directory	119
Viewing a File	120
Viewing the Properties of a File	120
Hardcopying a File	120
Renaming a File	120
Copying a File Into Another	121
Creating Links to Files	122
Deleting Files	122
Deleting Multiple Versions	122
Changing the Properties of a File	123
Creating a Directory	123
Buffer and File Attributes	125
Attributes	125
Attribute-Manipulating Commands	125
File Attribute Checking	126
Setting the Package	126
Other Set commands for File and Buffer Attributes	127
Dired Mode	131
Overview	131
Entering Dired	131
The Dired Display	131
Dired Commands	132
Default Pathnames in Dired	134
Getting Out of Dired	134
Online Documentation for Dired	135
Dired Menu	135
Moving Around in Dired	136
Viewing File Attributes in Dired	136

Changing File Properties in Dired	136
Viewing and Editing File Contents in Dired	137
Comparing Recent Versions of Files	137
Copying and Renaming Files	137
Marking Files for Deletion	138
Deleting Multiple Versions	138
Setting Generation Retention Count	139
Protecting Files From Being Reaped	139
Protecting Files From Being Deleted	140
Finding Files That Have Not Been Backed Up	140
Marking Files to be Hardcopied	140
Applying Arbitrary Functions to Files	141
<b>Setting the Major Mode</b>	<b>143</b>
Major Editing Modes	144
Overview	144
Fundamental Mode	144
Lisp Mode	144
Text Mode	144
Macsyma Mode	144
Midas Mode	145
Bolio Mode	145
Teco Mode	145
P11 Mode	145
Electric P11 Mode	145
<b>Changing Case and Indentation</b>	<b>147</b>
Changing Case	148
Overview	148
Changing Case of Words	148
Changing Case of Regions	148
Changing Case of Buffers	149
Indentation	150
Overview	150
Indenting Current Line	150
Centering the Current Line	150
Indenting New Line	151
Reindenting Expression	151
Indenting Region	151
Going Back to First Indented Character	151
Indenting Region Uniformly	152
Aligning Indentation	152
Deleting Indentation	152
New Line with This Indentation	152
Moving Rest of Line Down	153
Inserting Blank Line	153

Deleting Blank Line	153
<b>Editing Lisp Programs</b>	<b>155</b>
Introduction	156
Commenting Code	157
Overview	157
Indenting for Comment	157
Killing a Comment	157
Moving Down to Comment on Next Line	158
Moving Up to Comment on Previous Line	158
Setting the Comment Column	158
Creating a New Indented Comment Line	158
Commenting Regions	159
Evaluation and Compilation	160
Overview	160
Evaluation	160
Compilation	162
Compiler Warnings	163
Parenthesizing Expressions	164
Expanding Expressions	165
Locating Source Code to Edit	166
Introduction	166
The Edit Definition Commands	166
The List Definition Commands	168
The Edit Callers Commands	169
Patching	172
Introduction	172
The Patch Commands	172
<b>Customizing the Zmacs Environment</b>	<b>177</b>
Overview	178
Introduction	178
Built-in Customization — Zmacs Minor Modes	179
Definition — Minor Modes	179
How It Works	179
Summary of Minor Modes	180
Major Modes	182
User-Defined Major Modes	182
File Types and Major Modes	182
Creating New Commands: Keyboard Macros	183
Definition	183
How It Works	183
Procedure	183
Start Keyboard Macro	184
End Keyboard Macro	184
View Keyboard Macro	184

Call Last Keyboard Macro	184
Defining an Interactive Keyboard Macro	185
Naming a Keyboard Macro	185
Using Keyboard Macros to Sort	186
Installing a Macro on a Key	186
Installing a Mouse Macro	186
Deinstalling a Macro	186
More Features of the Keyboard Macro Facility	187
Key Bindings	191
Definition	191
How It Works: The Comtab	191
Set Key	191
Install Command	192
How to Specify Zmacs Variable Settings	193
Definition	193
Finding Out About Zmacs Variables	193
Describe Zmacs Variable	193
List Zmacs Variables	193
Variable Apropos	193
Set Variable	194
Customizing the Editor in Init Files	196
Introduction	196
Creating An Init File	196
Setting Editor Variables	196
Setting Mode Hooks	197
Key Bindings	198
<b>APPENDIX A. Help Command Summary</b>	<b>201</b>
Zmacs commands for finding out about the state of buffers	201
Zmacs commands for finding out about the state of Zmacs	201
Zmacs commands for finding out about Lisp	201
Zmacs commands for finding out about flavors	202
Zmacs commands for interacting with Lisp	202
<b>Index</b>	<b>203</b>

## Introduction

## Overview

---

### Scope

The *Zmacs Manual* is primarily a reference manual and is intended for all users of Zmacs on the Lisp Machine. It contains both conceptual overview and reference material that together describe the Zmacs editor. We assume that you have already read the *Lisp Machine Summary*.

---

### Organization

The first three chapters contain introductory material for users who are unfamiliar with Zmacs concepts. Experienced users can skim the remaining chapters, which are organized according to editing function, and use them as reference material.

"Introduction" gives an overview of Zmacs and describes Zmacs documentation conventions in this manual.

"Getting Started" introduces basic Zmacs concepts and commands, such as how to enter text, move the cursor, and make simple corrections.

"Getting Help" describes ways to get out of trouble and how to get Zmacs information during editing.

"Moving the Cursor" includes descriptions of both mouse and keyboard motion commands.

"Deleting and Transposing Text" explains Zmacs deletion and text retrieval concepts, as well as the ways to delete and transpose text.

"Working With Regions" tells how to manipulate blocks of text.

"Searching, Replacing, and Sorting" describes the commands for locating and replacing character strings in one or many files.

"Manipulating Buffers and Files" gives more information on manipulating blocks of text, inserting files, keeping track of everything, and editing your directory.



## Overview, cont'd.

---

"Setting the Major Mode" documents the major editing modes and their characteristics.

"Changing Case and Indentation" includes many commands for changing code, comments, or text to uppercase or lowercase, as well as commands for handling white space, indentation, and formatting.

"Editing Lisp Programs" the ways in which Zmacs is tailored for use in writing and editing programs in Lisp.

"Customizing the Zmacs Environment" describes how to fine tune your Zmacs environment using modes to set it up, keyboard macros to perform special editing tasks, binding keys to the commands of your choice, setting Zmacs variables to alter your standard system defaults, and saving the customized environment in init files.

Appendix A summarizes Zmacs help commands according to the context in which they are available.

---

## Introduction to Zmacs

---

### Overview

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

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

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

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

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

---

### Commands

Zmacs *commands* are Lisp functions that perform the editing work. Every Zmacs command has a *name*, and many commands are bound to keys. When a command is bound to a *keystroke combination*, you invoke it by pressing those keys. For example, the Forward Word command is invoked by typing the keystroke  $m-F$ . When a command is not bound to a set of keystrokes, Zmacs calls it an *extended* command and you invoke it using its name preceded by  $m-X$ . For example, the command View Mail, an extended command, is invoked View Mail ( $m-X$ ). *Command tables* assign keystrokes and names to commands. Each time you press a key, Zmacs looks up the function associated with that key. For ordinary characters, the function **com-standard**, in the standard command table, inserts the character once.

---

### Keystrokes

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

---

## Introduction to Zmacs, *cont'd.*

---

down the other shift keys, CONTROL, META, HYPER, and SUPER, determines the *modifier* part of the keystroke.

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

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

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

---

### Extended Commands

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

m-X Extended Command

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

Command completion is provided. (See "Completion" on page 12.)

---

### Command Tables

There is always a currently active command table (*comtab*). When you invoke a command, Zmacs looks it up in the associated command table, checks to see if it is valid in the current context, and performs the function. Zmacs uses many comtabs, a few of which are the standard comtab, a comtab for commands that begin with the c-X prefix, and a comtab for reading pathnames in the minibuffer.

Many commands have no meaning outside their own limited context. Sometimes you may get a message or see online documentation about a command that says  
Not available in current context. Those commands that are not accessible via a keystroke and not accessible via m-X are likely to be commands that do not work in the current context. For example, a command that is part of Dired is only available on a key when you are in Dired.

## Introduction to Zmacs, *cont'd.*

---

You can invoke a command that is not available in the current comtab with the `c-m-x` command. `c-m-x` works like `m-x`: you press the keys and then type the command name in the minibuffer. This is primarily intended for debugging new editor commands that have not yet been installed on any key. Using `c-m-x` to invoke a command that is not in the current comtab because it only works in some other context is a sure way to get into trouble.

`c-m-x`

Any Extended Command

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

Command completion is provided.

---

---

## Additional Notation Conventions

---

### Documentation

#### Conventions for Commands

The word *current*, when describing a word, line, paragraph, page, or any Zmacs-recognizable piece of text, refers to the text that currently contains (or immediately follows) the cursor.

The *invocation* of a command shows exactly what keys you must press to invoke, or call, a command. We use the following format to describe Zmacs commands:

invocation	Name
alternate invocation	
alternate invocation	

Formal description of command

Since each extended (m-x) command contains its name as part of its invocation, we do not repeat the name again on that line.

---

#### Example 1

m->	Goto End
-----	----------

Moves point to the end of the buffer.

With a numeric argument *n* between 0 and 10, moves point to a place *n*/10 of the way from the end of the buffer to the beginning.

(The m-> command goes to the end of the buffer — its name is Goto End.)

---

#### Example 2

Dired (m-x)

Prompts for the name of a directory to edit with Dired.

(The Dired (m-x) command is an extended command that enters the directory editor.)

---

#### Example 3

m-M	Back To Indentation
c-M-M	
m-RETURN	
c-M-RETURN	

Positions point before the first nonblank character on the current line.

(Back to Indentation has several possible invocations that all move back to the first nonblank character on the line.)

---



## Getting Started

## Entering Zmacs

---

### Introduction

You can enter, or invoke, the editor in several ways: Press SELECT E, use the mouse, or run either the function **(ed)** or the function **zwei:edit-functions**.

---

### 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.

---

### Mouse

You can invoke the editor using the mouse.

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

---

### (ed)

The Lisp function **ed** enters Zmacs from a Lisp Listener.

**ed** &optional *arg*

*Function*

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

*arg* can have these values.

*Value*

*Description*

t

The **ed** function enters the editor, creates an empty buffer, and selects it.

Pathname or string

The **ed** function enters the editor and finds or creates a buffer with the specified file in it.



## Entering Zmacs, *cont'd.*

---

Defined symbol

*(ed 'giraffe)*

The editor tries to find the source definition of that symbol for you to edit. A defined symbol can be, for example, a function, macro, variable, flavor, or system.

The symbol **zwei:reload**

The system reinitializes the editor. This destroys all existing buffers, so use this only if you have to.

---

### **zwei:edit-functions**

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

**zwei:edit-functions** *spec-list*

*Function*

**zwei:edit-functions** is like **ed** in that inside the editor process it throws you back into the editor, whereas from another process it just sends a message to the editor and selects the editor's window. **zwei:edit-functions** gives *spec-list* to the editor in the same way that Edit Callers (see page 169) and similar editor commands would.

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

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

---

## Getting Help in Zmacs

---

### Introduction

Zmacs has many features that provide information about Zmacs commands, existing code, buffers, and files. Two features are generally useful: the HELP key and completion. (See the chapter "Getting Help", page 32, for details.)

---

### HELP

Pressing the HELP key in a Zmacs editing window gives information about Zmacs commands and variables. (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193). The kind of information it displays depends on the key you press after HELP.

HELP ? <i>or help help</i>	Displays a summary of HELP options.
HELP A	Displays names, key bindings, and brief descriptions of commands whose names contain a string you specify. (The A refers to <i>apropos</i> , the name of the function that finds the commands and displays their documentation.)
HELP C	Displays the name and description of a command bound to a key you specify.
HELP D	Displays documentation for a command whose name you specify.
HELP L	Displays a listing of the last 60 keys you pressed.
HELP U	Offers to undo the last major Zmacs operation, such as sorting or filling, when possible.
HELP V	Displays the names and values of Zmacs variables whose names contain a string you specify. (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193).
HELP W	Displays the key binding for a command you specify. (The W refers to where.)
HELP SPACE	Repeats the last HELP 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.

## Getting Help in Zmacs, *cont'd.*

---

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

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

---

## Yanking

*Yanking* helps you to get back any text that you have typed or deleted, by retrieving it from a *history*. A history remembers commands and pieces of text, placing them in a *history list* in stack order, with the newer elements at the top of the history and the older elements toward the bottom. Yanking commands yank back an element of a history from any position in the history list that you specify:

Yanking in the kill history:

<code>c-0 c-Y</code>	Displays the elements of the kill history (saved text). Click left on ( <i>N</i> more elements in history.) to display those not shown.
<code>c-Y</code>	Yanks the first element in the kill history.
<code>m-Y</code>	After <code>c-Y</code> , yanks the previous element in the kill history. Subsequent <code>m-Y</code> s move down the kill history list.

---

**Getting Help in Zmacs, cont'd.**

---

**Yanking in the command history:**

- c-0 c-M-Y** Displays the elements of the command history (editor commands that use the minibuffer in any way). Click left on (*N* more elements in history.) to display those not shown.
- c-M-Y** Yanks the first element in the command history.
- m-Y** After **c-M-Y**, yanks the previous element in the command history. Subsequent **m-Y**s move down the command history list.

Killing and yanking are fully described in the chapter "Deleting and Transposing Text", page 60.

---

## Organization of the Screen

---

### Introduction

Zmacs divides its window into several areas — the editor window, the echo area, and the mode line, each of which contains its own type of information.

---

### Editor Window

The biggest area, the *editor window*, shows the text you are editing. You can edit several different items at once with Zmacs; each item is edited in a separate editing environment called a *buffer*.

---

### Buffer

Zmacs gives every buffer a name. At any time you are editing only one of them, the *selected* buffer. When we speak of what some command does to "the buffer", we are talking about the currently selected buffer. Multiple buffers in Zmacs make it easy to switch among several files; the mode line tells you which one you are editing.

---

### Cursor and Point

The small blinking rectangle, the *cursor*, usually appears somewhere within the buffer, showing the position of *point*, the location at which editing takes place. Although the cursor covers a single character, we consider point to be at the left edge of the cursor, between the character the cursor is blinking on and the previous character.

---

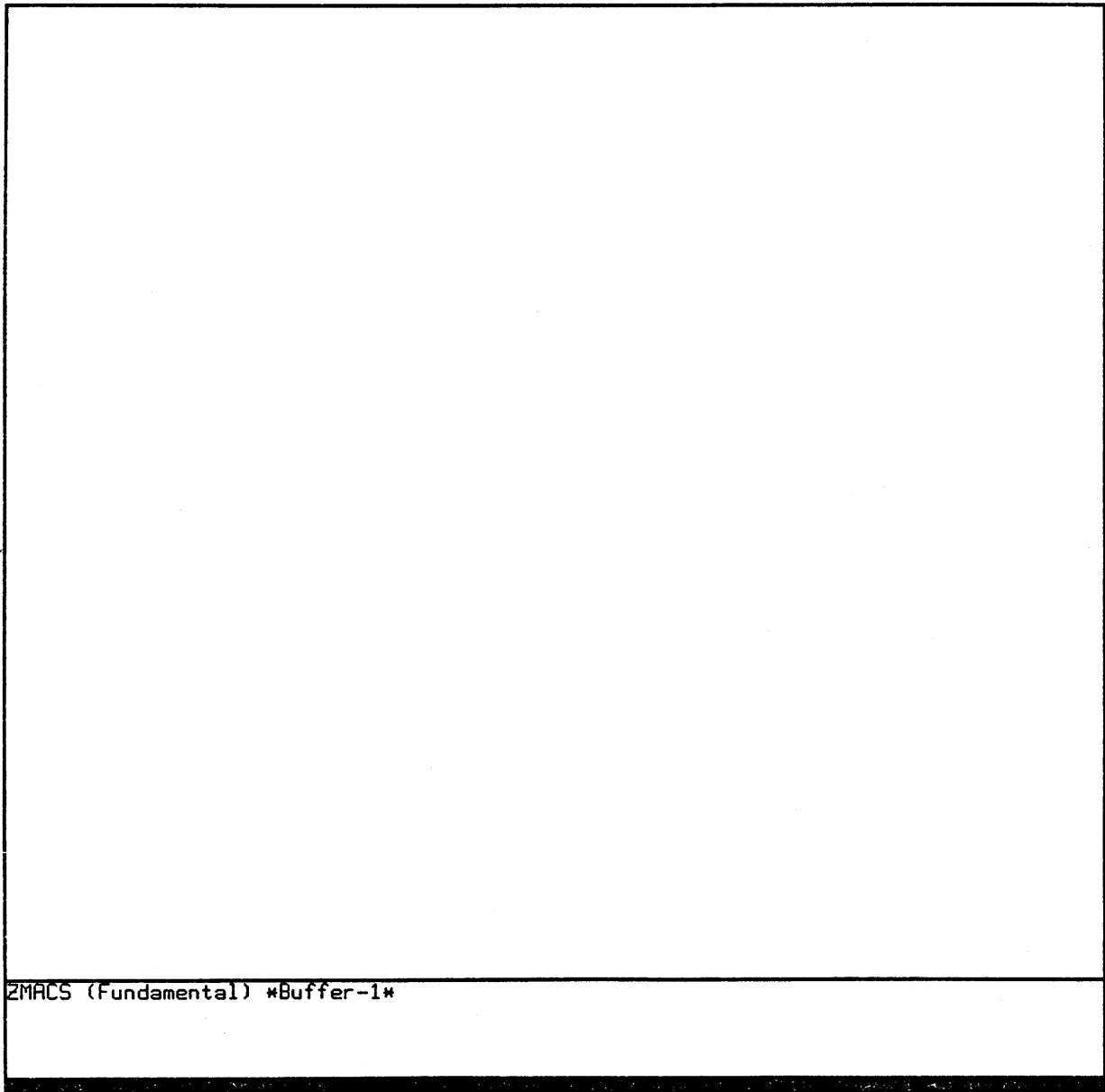
### Typeout

When you request some other information from Zmacs (for example, if you ask for HELP or a listing of a file directory), Zmacs needs room to display this type of information. It prints this *typeout* in a *typeout window* (at the top of the editor window), which temporarily overlays the text in the editor window, using as much room as it needs. Since the typeout is not part of the file you are editing, Zmacs delineates it from the editor buffer by drawing a line across the window between them (if both are present). The typeout window goes away if you type any command; if you want to make it go away immediately but not do anything else, you can press SPACE. The cursor, which appears at the end of the typeout, then moves back to its original location in the buffer.

---

**Organization of the Screen, cont'd.**

---



ZMACS (Fundamental) \*Buffer-1\*

## Organization of the Screen, *cont'd.*

---

### Echo Area

A few lines at the bottom of the screen make up what is called the *echo area*. *Echoing* means displaying the commands that you type. Zmacs commands are usually not echoed at all, but if you pause in the middle of a multicharacter command, then all the characters (including numeric arguments and prefixes) typed so far are echoed. This is intended to prompt you for the rest of the command. The rest of the command is echoed, too, as you type it. This behavior is designed to give confident users optimum response, while giving hesitant users maximum feedback.

---

### Minibuffer

Many Zmacs commands prompt you for additional information. This prompting happens in a small window within the echo area called the *minibuffer*.

When Zmacs prompts you, the cursor in the main editing window stops blinking and a blinking cursor appears in the minibuffer. Over the minibuffer, in the Zmacs mode line, some prompting text appears, indicating what information Zmacs is prompting you for.

When you type a response to the prompt, that response is inserted in the minibuffer. You can edit text in the minibuffer using the same Zmacs commands used in the main Zmacs window.

When you are done typing (and possibly editing) a response to the prompt, the RETURN key finishes your response.

---

### Mode Line

The line above the echo area is known as the *mode line*. It is the line that usually starts with ZMACS (Fundamental). Its purpose is to display information about the current buffer. The mode line consists of:

- The name of the major mode
- The name of the minor mode(s), if any
- The name of the buffer
- The version number of the file
- The status of the buffer
- A message telling whether the buffer contents extend above and/or below the screen

The mode line has this format:

```
ZMACS (major-mode minor-mode(s)) buffer (version) buffer-status  
[position-flag]
```

---

---

**Organization of the Screen, cont'd.**

---

*Major-mode*

*major-mode* is always the name of the *major mode* you are in. At any time, Zmacs is in one and only one of its possible major modes. The major modes available include:

- Fundamental mode (which Zmacs starts out in)
- Text mode
- Lisp mode
- Macsyma mode

See the chapter "Setting the Major Mode", page 141, for full details about all the major modes, how they differ, and how to select one.

---

*Minor-mode*

*minor-mode* is a list of the *minor modes* that are turned on at the moment. For example:

Fill	Auto Fill Mode
Electric Shift-lock	Electric Shift Lock Mode
Abbrev	Word Abbrev Mode
Overwrite	Overwrite Mode

See the chapter "Built-in Customization — Zmacs Minor Modes", page 179, for more information.

---

*Buffer*

*buffer* is the name of the workspace that holds the text you are editing. A buffer can be named in one of two ways:

- By Zmacs, with a name that corresponds to the existing file that it contains or with its standard name for an empty buffer
- By you, with any name you like

When a buffer contains a file, the buffer name is the pathname of that file, rearranged with the file name first and the host and directory at the end. (Pathname components are fully described in the *Lisp Machine Summary*.) When a buffer does not contain a file, the buffer name is a string. *whatever.lisp >wolf 4: (5)*

Buffers that do not contain files are empty, newly created, or temporary buffers. When Zmacs creates and names a buffer, that name begins and ends with an asterisk. When you create and name a buffer, on the other hand, its name is of your choosing.



## Organization of the Screen, *cont'd.*

---

When you first start up and enter Zmacs, your buffer is either:

- An empty buffer called \*Buffer-1\*, which is the only one that exists when Zmacs starts up
- A buffer containing an existing file, which Zmacs appropriately calls by its name

See the chapter "Multiple Buffers and Windows", page 103, for information on multiple buffers.

---

### *Version*

*(version)* is the version number most recently visited or saved. The mode line does not display any version number if the file is on a file system that does not support version numbers, such as UNIX.

... (5)

---

### *Buffer-status*

If the mode line displays \*, then changes have been made in the buffer that have not been saved in the file. If the buffer has not been changed since it was read in or saved, the mode line does not display a asterisk.

*after version #*

---

### *Position-flag*

When the mode line displays the message [More above], then your screen shows the end of your buffer contents; when the mode line shows [More below], then your screen shows the beginning of your buffer contents. When it says [More above and below], then the buffer contents extend above and below the part that the screen displays. When the display shows the entire buffer contents, this message does not appear at all.

---

### *Example*

```
ZMACS (Text) text.text /dess/doc/books/ VAX: * [More above and below]
```

In this sample mode line, we are in Zmacs Text Mode, editing a file named text.text, which resides in the directory /dess/doc/books on the host named VAX. The file has been changed since we last saved it (indicated by the \*), and the file contents extend above and below the portion that Zmacs displays on the screen.

---

---

## Inserting Text

---

### Introduction

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.

---

### Inserting Characters

When you type in new text, you are actually issuing Zmacs commands. Ordinary printing characters are called *self-inserting* because when you type one, it inserts itself into the text in your buffer.

You can give numeric arguments to the keystrokes that insert printing characters into the buffer; Zmacs interprets these arguments as repeat counts.

Example: `c-80 *` inserts a line of 80 asterisks at the cursor.

---

### Starting a New Line

*Newline* characters delimit lines of text. They have no visible printed form, but are present at each line break. You can break one line into two lines by inserting a newline (pressing RETURN) where desired. Similarly, you can merge two lines into one by deleting the intervening newline.

---

### Correcting Typos

To correct text you have just inserted, use the RUBOUT key. RUBOUT deletes the character *before* the cursor (not the one over which the cursor is positioned; that is the character *after* the cursor). The cursor and the rest of that line move to the left.

When the cursor is positioned on the first character on a line and you press RUBOUT, the preceding newline character is deleted and Zmacs appends the text on that line to the end of the previous line.

---

### Wrapping Lines

When you add too many characters to one line without breaking it with a RETURN, the line grows to occupy two (or more) lines on the screen, with an exclamation point at the extreme right margin of all but the last of them. The ! means that the following screen line is not really a distinct line in the file, but just the continuation of a line too long to fit the screen.

---

## Inserting Text, cont'd.

---

### Inserting Formatting Characters

You can insert most characters directly into the buffer by simply typing them, but other characters act as editing commands and do not insert themselves. If you need to insert a character that is normally a command (for example, **TAB** or **RUBOUT**), use the **c-Q** (Quoted Insert) command first to tell Zmacs to insert the following character into the buffer literally. **c-Q** prompts in the echo area for the character to be inserted and inserts it into the text.

---

? maybe not.  
I can't get  
it to work

---

## Numeric Arguments

---

### *Overview*

Many Zmacs commands take numeric arguments, which you type before the main command keystroke. Specify a numeric argument by pressing any combination of any of the modifier keys (c-, m-, s-, or h-) with the number. This way, you can type sequences of commands more easily without frequently alternating keys.

Numeric arguments to commands appear in the echo area when you do not type the command immediately. With no delay, the argument does not appear.

In general, use negative arguments to tell a command to move or act backwards. You can specify a negative argument by pressing any modifier key with the minus sign followed by the number. Most commands treat a numeric argument consisting of just a minus sign the same as -1.

---

### *Example*

c-F is the command to move the cursor forward one character. c-3 c-5 c-F moves point forward 35 characters; c-- c-3 c-5 c-F moves point backward 35 characters.

Throughout this manual, instead of writing out c-4 c-5 c-F or m-4 m-5 m-B, we will usually abbreviate to c-45F or m-45B.

---

### *Defaults*

Many commands have default numeric arguments. This means that in the absence of a numeric argument, the command behaves as if the default argument was given. Most commands have a default argument of 1. This includes all the commands that interpret numeric arguments as repeat counts. Some commands have a different default and still others have no default: their behavior in the absence of a numeric argument is different from their behavior with a numeric argument.

---

**c-U** Quadruple Numeric Arg  
This special command prefixes other commands, usually representing a numeric argument of 4. You can repeat c-U; it multiplies the numeric argument by 4 each time. For example, c-U c-U c-F moves point forward 16 characters. Sometimes instead of representing a numeric argument of 4, c-U alters the action of a command slightly; for example, when used with the command Set Pop Mark, described in "Working with Regions", page 72, c-U takes different actions with the mark.

---

## Moving the Cursor

---

### Description

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

The commands listed here and other cursor-moving commands are described in detail in the chapter "Moving the Cursor", page 44.

---

### Summary

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.	

---

**Moving the Cursor, cont'd.**

---

<b>c-X [</b> Moves to the previous page.	<b>Previous Page</b>
<b>c-V</b> Moves down to display the next screenful of text.	<b>Next Screen</b>
<b>m-V</b> Moves up to display the previous screenful of text.	<b>Previous Screen</b>
<b>m-&lt;</b> Moves to the beginning of the buffer.	<b>Goto Beginning</b>
<b>m-&gt;</b> Moves to the end of the buffer.	<b>Goto End</b>

---

## Erasing Text

---

### Description

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

If you issue a kill command by mistake, you can retrieve the text with `c-Y`, the Yank command. See the chapter "Working with Regions", page 72, for details on killing and retrieving text.

---

### Summary

<code>c-D</code> Deletes the character after point.	Delete Forward
RUBOUT Deletes the character before point.	Rubout
<code>m-D</code> Kills forward one word.	Kill Word
<code>m-RUBOUT</code> Kills backward one word.	Backward Kill Word
<code>m-K</code> Kills forward one sentence.	Kill Sentence
<code>c-X RUBOUT</code> Kills backward one sentence.	Backward Kill Sentence
<code>c-K</code> Kills to the end of the line or kills an end of line.	Kill Line
<code>c-W</code> Kills region (from point to mark).	Kill Region
<code>c-m-K</code> Kills forward over exactly one Lisp expression.	Kill Sexp
<code>c-m-RUBOUT</code> Kills backward over exactly one Lisp expression.	Backward Kill Sexp

---

**Erasing Text, cont'd.**

---

m-\	Delete Horizontal Space
Deletes any spaces or tabs around point.	
c-X c-0	Delete Blank Lines
Deletes any blank lines following the end of the current line.	
m-^	Delete Indentation
Deletes RETURN and any indentation at front of line.	

---



---

## Creating and Saving Buffers and Files

---

### Description

You do all your text editing in Zmacs *buffers*, which are temporary workspaces that can hold text. To keep any text permanently you must put it in a *file*. Files store data for any length of time.

To edit the contents of a file using Zmacs, you create a buffer and copy the file contents into it. To add text to the end of the buffer, move point to the end of the buffer and type the new text.

Editing proceeds in the buffer, not in the file. The file remains unchanged until you explicitly write the modified buffer contents to the file.

If you create multiple buffers, Zmacs keeps track of which files you are editing in which buffers. This association allows you to use completion to switch among buffers while you are editing them; you do not have to type the file name more than once. Zmacs always displays the name of the file you are currently editing.

The information in this section allows you to find or create and save a file; for complete information on buffers and files, see the chapter "Manipulating Buffers and Files", page 100.

---

### Summary

c-X c-F	Find File
Reads the specified file into a buffer.	
c-X c-S	Save File
Saves out the changes to the current file.	
c-X B	Select Buffer
Selects the specified buffer.	
c-X c-W	Write File
Writes out the buffer to the specified file.	

---

### Creating a Buffer

Zmacs creates your initial buffer when you first enter the editor.

To create other buffers, use c-X B, Select Buffer, to create an empty buffer or c-X c-F, Find File, to create either an empty buffer or a buffer containing a file.

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

---

## Creating and Saving Buffers and Files, *cont'd.*

---

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

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

<i>System</i>	<i>Empty Buffer Name</i>
Lisp Machine	foo.lisp
UNIX	foo.l
VMS	foo.lsp

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

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

---

### Saving a File

Once you have the file in your buffer, you can make changes and then *save* the file with `c-X c-S`, the Save File command. This makes the changes permanent and actually changes the file. Until then, the changes are only inside your Zmacs buffer and the file itself is not really changed.

---

### Creating a File

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

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

## Creating and Saving Buffers and Files, *cont'd.*

---

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

---

### Editing Existing Files

To tell Zmacs to edit text in a file, use `c-X c-F`, the Find File command, and give Zmacs a file name. You can enter the *pathname* of any file on any host that is reachable by network connections from your Lisp Machine. If the file already exists, Zmacs locates the file and reads it into your buffer.

---

## Leaving Zmacs

---

### Overview

Use a system-wide command to switch programs, such as SELECT, FUNCTION S, the System menu, or, if you have multiple windows on the screen, position the mouse to another window and click.

---

### SELECT Key

A set of windows is always available by pressing the SELECT key and then one of the following keys:

<i>Key</i>	<i>Program</i>
C	Converse, for messages to other users
E	Editor, the Zmacs text and program editor
F	File system editor for access to files and directories
I	Inspector, for inspecting and modifying data structures
L	Lisp
M	Mail reading and sending system
P	Peek, a system status display
T	Telnet, a virtual terminal utility for logging in to other hosts
X	Favor Examiner, for examining the structure of flavors that are defined in the Lisp environment

---

### System Menu

The System menu is a momentary menu that lists several choices for acting upon windows and calling programs (for example, a Lisp Listener, Zmacs, or the Inspector). You can always call the System menu by clicking [(R2)] (the right mouse button twice or holding down the SHIFT key and clicking right once). 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.
- 

### c-z

The Zmacs command c-z returns you to the window in which the (ed) function was most recently called, usually the Lisp Listener.

---

## Getting Help

## Getting Out of Trouble

---

### Overview

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.

---

### Getting out of Prefixes and Prompts

Most of the commands we have described are single keystrokes, but some keystrokes are prefixes that must be completed with a second keystroke to specify a command. `c-X` is the most important of these.

---

#### *Prefixes*

If you press a `c-X` and don't mean it, you can get out by pressing either `c-G` or `ABORT`. These are general "get me out of here" commands, which you should use whenever you get yourself into a confused state. `ABORT` and `c-G` are, for the most part, synonymous in Zmacs.

---

#### *Minibuffer Prompts*

Sometimes you accidentally type a command that prompts for some additional information, or you type such a command on purpose and change your mind afterwards. When Zmacs prompts and you just want to get out of the minibuffer and back to where you were, press `ABORT`. If, instead, you wish to cancel and reenter your response, use `c-G`, which clears any typein but leaves you still in the minibuffer. When the minibuffer is empty, `c-G` cancels the minibuffer command. (With some echo area prompts, you have to use `ABORT`.)

---

`ABORT`

Abort At Top Level

Cancels the last command typed. It also cancels numeric arguments and region marking.

---

`c-G`

Beep

Cancels the last command. It also cancels numeric arguments, and region marking, except when given an argument. It cancels one thing at a time, so that if you've typed a number of commands or

---

## Getting Out of Trouble, *cont'd.*

---

responses, you must use successive `c-Gs` to cancel each one and return to top level.

---

### 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. These delete and kill commands are fully described in the chapter "Deleting and Transposing Words", page 60.

---

### 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.

Killing and yanking is fully described in the chapter "Working with Regions", page 72.

---

---

## Finding Out About Zmacs Commands

---

### Overview

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.

---

### HELP

The HELP key is a prefix to a useful group of commands giving various kinds of online help. If you forget what a command does, which keystrokes perform an action, or have no idea how to accomplish something, press HELP.

Whenever you have a question of any kind, press HELP — Zmacs prompts you in the minibuffer for details on what kind of help. If you don't know, press HELP again and it tells you, in the *typeout window*, how to find what you're looking for. The typeout window displays right over the editor window. The actual contents of the buffer are not affected, and the next command you type restores the buffer display.

---

### Finding Out What a Command Does

#### HELP C

The command HELP C displays "Document Command:" below the mode line and waits for you to type a command. When you do, Zmacs displays the internal documentation for that command.

---

#### *Example*

If you press HELP-C followed by c-F, the response is:

```
c-F is Forward, implemented by COM-FORWARD:
Moves forward one character.
With a numeric argument (n), it moves forward n characters.
```

The first line above tells you the name of the command (in this case Forward), and the name of the internal Lisp function that actually does the work (in this case **com-forward**). (You don't need to know these internal names for basic editing.) The next line is a very short description of what the command does; it usually tells you what the command does without a numeric argument and how a numeric argument modifies that behavior.

---



---

## Finding Out About Zmacs Commands, *cont'd.*

---

### *Prefix Commands*

When you ask (with `HELP C`) for documentation on a prefix command like `c-x`, Zmacs prompts you, in the typeout window, to complete the command. Zmacs displays the documentation for the prefix command in the typeout window.

---

### *Extended Commands*

#### HELP D

When you want to find out what an extended command does, you can display the documentation for the command by pressing `HELP D`, which prompts in the minibuffer "Describe command:", to which you type the command's name.

---

### Searching for Appropriate Commands

#### HELP A

When you can only guess at part of the name of a command by the action it performs, there is a command, `HELP A`, to help you scan all the available Zmacs commands to find the one you want.

Each Zmacs command has a name. The name is almost always exactly what you would expect; that is, the name describes the function of the command in reasonably plain English.

---

### *Example*

The command you perform when you use `m-Q` is called "Fill Paragraph", so you might expect a command that counts the number of paragraphs in the buffer to be called something like "Count Paragraphs" or "Paragraphs Count". No matter what, the name is going to have the word *paragraph* in it.

---

### *How It Works*

To find the command you want, just press `HELP A`. Zmacs prompts you for a substring, you enter your guess, and then Zmacs displays short descriptions of all the commands whose names contain that substring. If the string that you enter contains a space, then Zmacs displays a short description of all the commands whose names include a similarly positioned space. Each description gives the short documentation for the command and tells what keystrokes invoke it.

---

---

## Finding Out About Zmacs Commands, *cont'd.*

---

### Finding Out What You Have Typed

#### HELP L

As you are editing you might find yourself in a hopelessly confused state and not know how to recover.

If this happens to you it is often very enlightening to press HELP L to list the last 60 keystrokes you typed. By examining your own recent activity, it is often possible to find out where you went wrong and how to save yourself.

---

### More HELP Commands

#### HELP U

Offers to undo the last "major" operation (such as fill or sort).

---

#### HELP V

Displays all the Zmacs variables whose names contain a certain substring. (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193).

---

#### HELP W

Finds out whether an extended command is bound to a key.

---

### General Information-giving Commands

The following commands display:

- Information about the location of point
- Documentation about a specified Lisp function
- Argument list for the specified Lisp function
- Information about the current Lisp variable
- The number of lines in the region or page
- Possible parenthesis mismatches
- Trace information about the specified Lisp function

The word *current*, when describing a Lisp function or a Lisp variable, refers to (approximately) the function or variable whose name is nearest to the cursor.

---

---

## Finding Out About Zmacs Commands, *cont'd.*

---

### *About Point*

`c-X =` Where Am I

Displays various things about where point is. It displays the X and Y positions, the octal code for the following character, the current line number and its percentage of the total file size. If there is a region, it displays the number of lines in it. Fast Where Am I (`c-=`) displays a subset of this information faster.

---

`c-=` Fast Where Am I

Quickly displays various things about where point is. It displays the X and Y positions and the octal code for the following character. If there is a region, it displays the number of lines in it. Where Am I displays the same things and more.

---

### *About Function Documentation*

`c-sh-D` Brief Documentation

Displays brief documentation for the specified Lisp function. By default, it displays documentation for the current function. With a numeric argument, it prompts for a function name, which you can either type in or select with the mouse. It displays the first line from the summary paragraph in the echo area.

---

`m-sh-D` Long Documentation

Displays the documentation for the specified function. It prompts for a function name, which you can either type in or select with the mouse. The default is the current function.

---

### *About Displaying Argument Lists*

`c-sh-A` Quick Arglist

Displays the argument list for the current function. With a numeric argument, it reads the function name from the minibuffer.

---

Arglist (`m-X`)

Displays the argument list of the specified function. It reads the name of the function from the minibuffer) and displays the argument list in the echo area.

---

---

**Finding Out About Zmacs Commands, cont'd.**

---

*About Describing  
Lisp Variables***c-sh-V** Describe Variable At Point

Displays information in the echo area about the current Lisp variable. The information displayed shows whether it is declared special, whether it has a value, and whether it has documentation put on by **defvar**. When nothing is available, it checks for lookalike symbols in other packages.

---

*About Counting  
Number of Lines***m-=** Count Lines Region

Displays the number of lines in the region.

---

**c-X L** Count Lines Page

Displays the number of lines on the current page (or the buffer, if there are no page delimiters). In parentheses, it displays the number of lines up to the line containing point and the number of lines after the line containing point.

---

*About Finding  
Unbalanced Parentheses***Find Unbalanced Parentheses (m-X)**

Finds any parenthesis mismatch error in the buffer. It reads through all of the current buffer and tries to find places in which the parentheses do not balance. It positions point to possible trouble spots, printing out a message that says what the trouble appears to be. This command only finds one such error; if you suspect more errors, run it again.

---

*About Tracing  
Function Executions***Trace (m-X)**

Traces or untraces a function. It reads the name of the function from the minibuffer and then it pops up a menu of trace options. With an argument, it omits the menu step.

---

## The Editor Menu

---

### Overview

Click right in Zmacs to display the *editor menu*, a momentary menu containing editor commands, each of which is a possible choice. Position the mouse cursor over an item and then click the appropriate button to make the choice.

The editor menu command summaries below point to complete descriptions in appropriate chapters of the manual.

---

### Editor Menu Commands

The Editor Menu commands are:

<i>Command</i>	<i>Description</i>
Arglist	Prints the argument list of the specified function (see page 37).
Edit Definition	Prepares to edit the definition of a specified function (see the chapter "Editing Lisp Programs", page 153).
List Callers	Lists all functions that call the specified function (see the chapter "Editing Lisp Programs", page 153).
List Definitions	Displays the definitions in a specified buffer (see the chapter "Editing Lisp Programs", page 153).
List Buffers	Prints a list of all the buffers and their associated files (see the chapter "Manipulating Buffers and Files", page 100).
Kill Or Save Buffers	Offers a menu of modified files with choices to kill, save, or remove the modification flag from the file (see the chapter "Manipulating Buffers and Files", page 100).
Split Screen	Makes several windows split among the buffers as specified (see the chapter "Manipulating Buffers and Files", page 100).
Compile Region	Compiles the region, or if no region is defined, the current definition (see the chapter "Editing Lisp Programs", page 153).

---

**The Editor Menu, *cont'd.***

---

Indent Region	Indents each line in the region (see the chapter "Changing Case and Indentation", page 145).
Change Default Font	Sets the default font (see the chapter "Working With Regions", page 72).
Change Font Region	Changes the font for the region (see the chapter "Working With Regions", page 72).
Uppercase Region	Changes any lowercase characters in the region to uppercase (see the chapter "Working With Regions", page 72).
Lowercase Region	Changes any uppercase characters in the region to lowercase (see the chapter "Working With Regions", page 72).
Indent Rigidly	Shifts text in the region sideways as a unit (see the chapter "Changing Case and Indentation", page 145).
Indent Under	Indents to align under a string read from the minibuffer (see the chapter "Changing Case and Indentation", page 145).

---

---

## More on the Minibuffer

---

### Response Format

Most commands only expect one line of response. In these cases, the END key has the same meaning as the RETURN key, terminating the response. (In completion, the RETURN key is not exactly the same as the END key — see below.)

However, for commands that expect one or more lines of response, RETURN has its usual significance, inserting a newline in the minibuffer, and END marks the end of the response.

---

### Response Help

While responding to a prompt, you can press HELP to get documentation describing the current situation. Zmacs tells you exactly what input it expects and what the possible responses are.

---

### More Ways to Enter Responses

Yanking and mousing provide quick and simple ways to enter minibuffer responses without having to type them out. Both of these methods are context-sensitive. Yanking works only when you have previously entered a minibuffer response. Mousing works when you click on a name that makes sense in the context of the minibuffer prompt.

---

### *Yanking*

**c-m-Y** Repeat Last Minibuffer Command

Repeats a recent minibuffer command. It yanks the displayed default if there is one, otherwise, it yanks the last thing typed in this context. A numeric argument  $n$  yanks the  $n$ th previous one. An argument of 0 lists the history of elements typed in the minibuffer.

---

**m-Y** Repeat Last Minibuffer Command

After **c-m-Y**, **m-Y** replaces what was yanked with a previous element of the same history. A numeric argument of zero displays the history. A positive numeric argument moves to that much older a history element. A negative numeric argument moves to a newer history element; this only makes sense after the history has been rotated.

For more details, see "Retrieving History Elements", page 64.

---

---

**More on the Minibuffer, cont'd.**

---

**Mousing**

If the mouse is an arrow pointing straight up, you can point at the name of something (for example, a function if the command is reading a function name in the minibuffer) and click the left button. Mouse-sensitive things that could be a valid argument are highlighted with a box. The mouse only works this way when the minibuffer is empty. If you type something and then decide that you would rather use the mouse, erase what you typed with RUBOUT or CLEAR-INPUT.

---

**Completion**

Sometimes, when a command prompts you, you have only a limited number of possible responses. The responses themselves can be cumbersome to type. To save you from having to type the entire response, some commands perform *command completion*.

*Completing* means presuming the rest of your response, based on what you have typed already. Each command that offers completion has a list of acceptable answers and it checks what you have typed so far against the list.

When Zmacs is reading a command argument from the minibuffer and some sort of command or file name completion is available, the right-hand side of the mode line says (Completion). You will soon acquire a feeling for the contexts in which Zmacs provides completion.

---

**Completion Commands**

The commands described in this section only behave in the indicated manner when completion is allowed.

---

**COMPLETE****Complete**

Pressing the COMPLETE key asks Zmacs to try to complete the response you have typed so far.

Three things could happen:

1. In the optimal case, the response you have typed so far will have exactly one completion. In this case, Zmacs performs the completion. You can then press END to terminate the response and continue the execution of the prompting command. Or, you can choose to continue editing the response.
2. Often you will find that you have not yet typed enough to specify a valid response unambiguously. When there is more than one valid completion, Zmacs completes as far as it can and



---

## More on the Minibuffer, *cont'd.*

---

then waits for more input from you since your response is not yet complete. You can then complete your response by typing more letters to clearly specify your desired response, thereby disqualifying any other valid ones.

3. In the worst case, the response you have typed so far has no valid completion. In this case, Zmacs beeps (audibly on the LM-2) and continues to wait for additional input in the minibuffer. You can continue to edit your response.

---

### END Complete And Exit If Unique

Pressing the END key tries to complete your response so far. If the completion is successful, it terminates the response and continues executing the prompting command. If the completion is unsuccessful (if the response was ambiguous or cannot be completed in its present form), Zmacs waits for you to continue editing it.

---

### *Impossible-is-OK Completion*

Each command that provides completion has a list of valid responses. These are not always the *only* possible responses: It might make sense for you to type a response the command had never heard of. When this is true, the command does a special kind of completion called *impossible-is-OK* completion. This is implemented with the RETURN key.

---

### RETURN Complete And Exit

Pressing the RETURN key tries to complete your response so far. If we are doing impossible-is-OK completion, RETURN terminates the response and returns to the prompting command *whether or not the completion was successful*. Otherwise, it behaves exactly like END.

---

### *Completing Responses in Chunks*

Often the desired response has several components separated by spaces or punctuation marks (for example, parentheses or hyphens). The components are called *chunks*. Zmacs, rather than always trying to complete the response as a unit, completes all the chunks separately and in parallel. For example, `co b` completes to `Compile Buffer` in spite of other possible completions of `co`, such as `Copy File` and `Count Lines`. When the response is ambiguous, Zmacs completes the chunks that it can and positions the minibuffer's cursor at the leftmost chunk that needs further clarification.

---

**More on the Minibuffer, *cont'd.***


---

SPACE Self Insert and Complete  
)

When you press the SPACE bar, a close parenthesis, or any chunk delimiter (chunk delimiters are context-dependent) you have finished typing one chunk of your response. Zmacs then tries to complete that chunk as part of the command name. If it does not succeed, it assumes that you are not finished specifying your entire response. If at any point it cannot supply a possible completion, it beeps.

*Example*


---

The following command completes to Source Compare instead of to Source Compare Merge:

m-X so SPACE co SPACE RETURN

The following commands complete to Source Compare Merge:

m-X so SPACE co SPACE m END

m-X so SPACE co SPACE m RETURN

---

*Enumerating  
Possible Completions*

c-? List Completions  
[Mouse (R)]

Enumerates the possible completions of your response so far. Zmacs lists the possible completions in the typeout window. The completions are mouse-sensitive, so you can select one by pointing at it with the mouse and clicking left. [Mouse (R)] pops up a menu, which also lists the possible completions.

---

c-/ Completion Apropos

Enumerates the commands whose names contain the response as a substring. The command names are mouse-sensitive and you can select one by clicking on it.

---

*Getting Help While Completing*

When completion is provided, the HELP key provides a summary of the completion commands and a mouse-sensitive list of possible completions, in addition to the standard documentation for whatever command is prompting you.

---

## **Moving the Cursor**

## Overview

---

### Introduction

To make changes at some particular place in a Zmacs buffer, you must move the cursor to that place, since most commands that modify the buffer do so immediately around the cursor.

This section describes the commands that:

- View the contents of the buffer
  - Redisplay the editor window
  - Move the cursor around the buffer using mouse commands
  - Move the cursor around the buffer using keystroke commands
- 

### The Editor Window

The *editor window* displays either a portion of your buffer or the whole buffer, depending on the size of the buffer and your current location in it.

When the current buffer is smaller than the exact size of the editor window, Zmacs displays the contents of the buffer at the top of the window and leaves the bottom of the window blank. You cannot tell whether the buffer actually comes to an end where the text stops, since there could be white space and newline characters after the last visible piece of text.

When the buffer is too large to fit on the screen, the editor window shows only a section of the buffer. The part that shows always contains the cursor, so it never vanishes off the top or bottom of the editor window. Zmacs changes the position of the editor window inside the buffer as seldom as possible — usually only when you try to move the cursor off the top or bottom of the screen.

---

### *Wraparound Lines*

Lines that are too long to fit across the editor window are displayed on as many physical lines as are necessary. An exclamation point (!) in the (normally blank) last column means that the next physical line is part of the same logical line.

---

---

## Redisplaying the Window

---

### Introduction

Whenever you modify the buffer's contents or move point or the mark (see the chapter "Working With Regions", page 74, for a discussion of the mark), Zmacs updates the display to reflect the change. This updating can be as simple as moving the cursor or as involved as figuring out the whole display from scratch. These operations are called *redisplay* and Zmacs performs them automatically.

For example, when you move the cursor off the top or bottom of the editor window, a complete redisplay is required. The window has to shift to show a different part of the buffer in order to keep the cursor visible.

You can explicitly tell Zmacs to do a redisplay with the Recenter Window command, invoked by `c-L`. You might want to do this if the cursor gets too close to the top or the bottom of the editor window, and you want to redisplay with the cursor closer to the center so that you can see more context in one direction or the other.

It is important to remember that redisplay operations change only the *display*, not the actual contents of the buffer.

---

### Recentering Window

`c-L` Recenter Window

Completely redisplay the screen, leaving the cursor near the middle of the editor window.

With a positive numeric argument of  $n$ , it leaves the cursor  $n$  lines from the top of the window. With a negative numeric argument of  $-n$ , it leaves the cursor  $n$  lines from the bottom of the window.

---

### Next Screen

`c-V` Next Screen

Moves the cursor to the beginning of the last visible line in the editor window and redisplay the screen with that line at the top of the window.

With a numeric argument of  $n$ , it moves the text up  $n$  lines. With a negative numeric argument  $-n$ , it moves the text down  $n$  lines. The cursor does not move (with respect to the text) unless the numeric argument is large enough to slide it off the screen. In that case the cursor remains at the top.

---

---

**Redisplaying the Window, cont'd.**

---

**Previous Screen**

M-V

Previous Screen

Moves the cursor to the beginning of the first visible line in the editor window and redisplay the screen with that line at the bottom of the window.

With a numeric argument of  $n$ , it moves the text down  $n$  lines. With a negative numeric argument  $-n$ , it moves the text up  $n$  lines. The cursor does not move (with respect to the text) unless the numeric argument is large enough to slide it off the screen. In that case the cursor remains at the bottom.

---

**Positioning  
Window Around Definition**

C-M-R

Reposition Window

Redisplays, trying to get all of the current function definition in the window. It puts the beginning of the current definition at the top of the window with the current position of the cursor still visible. Doing C-M-R twice pushes comments off the top of the window, making more of the code of a large function visible.

---

**Moving to  
Specified Line**

M-R

Move To Screen Edge

Moves to the beginning of a specified line on the screen. With no argument, it moves to the beginning of a line near the middle of the screen. The exact line is controlled by the Zmacs variable Center Fraction. A numeric argument specifies a particular line to move to. Negative arguments count up from the bottom of the window. (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193).

---

## Using the Mouse

---

### Introduction

The easiest way to get the cursor where you want it is with the *mouse*. (The mouse is fully documented in the *Lisp Machine Summary*.)

---

### Mouse

#### Documentation Line

The mouse documentation line:

- Appears just above the bottom line of the screen
- Normally stands out in reverse video
- Contains documentation on the current meaning of mouse clicks

In a regular Zmacs buffer, the mouse documentation line offers the following options:

<i>Notation</i>	<i>Description</i>
L:Move point	Performs two separate actions: <ul style="list-style-type: none"><li>• Relocates the cursor: position the mouse cursor to the desired location and click left.</li><li>• Makes a region: position mouse cursor to desired location, click left (keeping the button down), move mouse cursor to end of region and lift the button up.</li></ul>
L2:Move to point	Relocates the mouse cursor near the cursor: click left twice.
M:Mark thing	Marks a small region: position mouse cursor on either side or in the middle of a word, Lisp expression, or after the end of a line, and click middle. ( <i>Marking regions is fully described in the chapter "Working with Regions", page 72.</i> )
M2:Save/Kill/Yank	Performs one of four related actions: <ul style="list-style-type: none"><li>• If there is a region, it saves the region in the kill history while leaving it in the buffer (like <i>m-W</i>)</li><li>• If the last command saved the region, it wipes it from the buffer (like <i>c-W</i> except it does not save)</li><li>• If the above two conditions do not apply, it yanks the first element from the kill history (like <i>c-Y</i>)</li><li>• If the last command was a yank command, it yanks the next item from the kill history (like <i>m-Y</i>)</li></ul>

---

**Using the Mouse, cont'd.**

---

(*Saving, killing, and yanking* regions is fully described in the chapter "Working with Regions", page 72.)

R:Menu

Displays a Zmacs menu offering mouse-sensitive Zmacs commands.

R2:System Menu

Displays a System menu.

---



---

## Motion Commands

---

### Introduction

Zmacs word, sentence, and paragraph motion commands all have strict definitions for where words, sentences, and paragraphs begin and end. These definitions can all be modified by the user.

---

### Numeric Arguments

All of the motion commands allow numeric arguments. For the most part, these numeric arguments are interpreted as repeat counts.

---

#### *Example*

`m-F` moves the cursor forward one word, whereas `m-13F` moves the cursor forward 13 words.

---

### *Negative Numeric Arguments*

Most of the motion commands come in pairs, with one command for forward motion over a particular unit and one command for backward motion. Both kinds of commands often interpret negative numeric arguments by reversing the direction of motion.

These conventions — that Zmacs interprets numeric arguments as repeat counts, and that negative numeric arguments reverse the direction of motion — together make up the *motion convention*.

---

#### *Example*

`m- -13F` moves point backward 13 words. `m-13B` has exactly the same effect.

---

### Motion by Character

A Zmacs *character* can be any letter, number, or punctuation character.

---

#### *Forward Character*

`c-F`

Forward

Moves the cursor forward over one character. `c-F` interprets numeric arguments as repeat counts.

Negative numeric arguments reverse the direction of motion. For example, `c-3B` and `c- -3F` both move the cursor backwards three characters.

---

**Motion Commands, cont'd.**

---

*Backward Character*

**c-B** **Backward**

Moves the cursor backward over one character. c-B interprets numeric arguments as repeat counts.

Negative numeric arguments reverse the direction of motion. For example, c-3 c-B and c-- c-3 c-F both move the cursor backwards three characters.

---

**Motion by Word**

Zmacs generally considers a *word* to consist of a sequential string of alphanumeric characters, that is, any combination of the characters a-z, A-Z, and 0-9. Different major modes define their own delimiter characters. For example, in Text Mode an apostrophe (') is part of a word, but in other modes it is a delimiter (see the chapter "Setting the Major Mode", page 141, for mode descriptions).

---

*Forward Word*

**m-F** **Forward Word**

Moves the cursor forward one word. Numeric arguments are interpreted as repeat counts; negative numeric arguments reverse the direction of motion.

m-F always places the cursor at the end of a word. If the cursor is in the middle of a word, m-F moves the cursor to the end of that word.

---

*Backward Word*

**m-B** **Backward Word**

Moves the cursor backward one word. Numeric arguments are interpreted as repeat counts; negative numeric arguments reverse the direction of motion.

m-B always places the cursor at the beginning of a word. If the cursor is in the middle of a word, m-B moves the cursor to the beginning of that word.

---

**Motion by Sentence**

According to Zmacs, sentences can end with question marks, periods, and exclamation points. Furthermore, these punctuation marks only end a sentence when followed by:

1. a newline
2. a space followed by either a newline or another space.

---

## Motion Commands, *cont'd.*

---

However, Zmacs allows any number of *closing characters*, which are ", ', ), and ], between the sentence-ending punctuation and the white space that follows it. A sentence also starts after a blank line.

This corresponds pretty closely to standard typing conventions. Zmacs does not recognize a period followed by one space as the end of a sentence, for example, as in "e.g." or "Dr."

---

### *Forward Sentence*

m-E Forward Sentence

Moves the cursor forward one sentence.

Numeric arguments are interpreted as repeat counts; negative numeric arguments reverse the direction of motion.

m-E always places the cursor at the end of a sentence. If the cursor is in the middle of a sentence, m-E moves the cursor to the end of that sentence.

---

### *Backward Sentence*

m-A Backward Sentence

Moves the cursor backward one sentence.

Numeric arguments are interpreted as repeat counts; negative numeric arguments reverse the direction of motion.

m-A always places the cursor at the beginning of a sentence. If the cursor is in the middle of a sentence, m-A moves the cursor to the beginning of that sentence.

---

## Motion by Lisp Expression

The next several pages deal with moving the cursor according to Lisp code delimiters: *lists* and *expressions*. A list is something enclosed in balanced parentheses. A Lisp expression is any readable printed representation of a Lisp object — a list or the printed representation of an atom.

c-m-N Forward List

Moves forward over one list. It accepts a numeric argument for repetition count.

---

**Motion Commands**, *cont'd.*


---

**c-m-P** **Backward List**

Moves backward over one list. It accepts a numeric argument for repetition count.

---

*Motion Along One  
Nesting Level*

Point always sits either between two expressions or in the middle of an atom.

---

**c-m-F** **Forward Sexp**

Moves point to the end of a surrounding atom if there is one, or past the Lisp expression immediately to the right if not.

If parentheses are unbalanced to such an extent that it doesn't make sense to talk about "the expression on the right", this command gives an error message and does not move point at all.

**c-m-F** observes the motion convention for numeric arguments.

---

**c-m-B** **Backward Sexp**

Moves point to the beginning of a surrounding atom if there is one, or to the beginning of the Lisp expression immediately to the left if not.

If parentheses are unbalanced to such an extent that it doesn't make sense to talk about "the expression on the left", this command gives an error message and does not move point at all.

**c-m-B** observes the motion convention for numeric arguments.

---

*Motion Up and  
Down Nesting Levels*

**c-m-D** **Down List**

Moves point forward past any intervening atoms to the next nonatomic expression and leaves point just to the right of the open parenthesis of that expression.

With a numeric argument of  $n$ , it moves down  $n$  nesting levels.

---

---

## Motion Commands, *cont'd.*

---

c-m-U Backward Up List

c-m-(

Backs up out of nesting levels. It moves backward one level of list structure. It searches for an open parenthesis and leaves point to the left of that open parenthesis. Also, if called inside of a string, it moves back up out of that string, leaving point to the left of its starting quote. It accepts numeric arguments for repetition count.

With a numeric argument of  $n$ , it moves up  $n$  nesting levels.

---

c-m-) Forward Up List

Moves forward out of nesting levels. It moves forward one level of list structure. It searches for a close parenthesis and leaves point to the right of that close parenthesis. Also, if called inside of a string, it moves up out of that string, leaving point to the right of its ending quote. It accepts numeric arguments for repetition count.

With a numeric argument of  $n$ , it moves up  $n$  nesting levels.

---

### *Motion Among Top-Level Expressions*

A Lisp file contains a sequence of expressions that we call *top-level expressions*, to distinguish them from their own subexpressions. Zmacs assumes that top-level expressions begin with an open parenthesis against the left margin. It does *not* parse top-level expressions by balancing parentheses, since parentheses do not always balance while programs are being written. The indentation represents the *programmer's* conception of program structure, and provides a better guide. So by *top-level expression*, we mean a section of text delimited by open parentheses at the beginning of two lines.

In code that includes a string containing a carriage return followed by an open parenthesis, show that the open parenthesis does not start a top-level expression by putting a slash in front of it.

---

c-m-A Beginning Of Definition

c-m-[

Moves point to the beginning of the current top-level expression.

With a positive numeric argument  $n$ , it moves back  $n$  top-level expressions. With a negative numeric argument  $-n$ , it moves forward  $n$  top-level expressions.

---

**Motion Commands, cont'd.**


---

c-M-E End Of Definition

c-M-]

Moves point to the end of the current top-level expression.

With a positive numeric argument  $n$ , it moves forward  $n$  top-level expressions. With a negative numeric argument  $-n$ , it moves back  $n$  top-level expressions.

---

m-) Move Over )

Moves past the next close parenthesis, then does Indent New Line. It removes any whitespace between point and the close parenthesis before moving over it. With a positive argument  $n$ , after finding the next close parenthesis and deleting whitespace before it, it moves past  $n-1$  additional close parentheses before doing Indent New Line. It ignores numeric arguments that are less than 1.

---

**Motion by Line**

Lines are delimited by special characters called *newlines*.

---

*Down Line*

c-N Down Real Line

Moves the cursor straight down to the corresponding column of the next line. If the cursor is positioned in the middle of the line, c-N moves it to the middle of the next one.

With a numeric argument  $n$ , it moves the cursor down  $n$  lines. Moving down a negative number of lines is the same as moving up.

---

*Up Line*

c-P Up Real Line

Moves the cursor straight up to the corresponding column of the previous line. If the cursor is positioned in the middle of the line, c-P moves it to the middle of the previous one.

With a numeric argument of  $n$ , it moves the cursor up  $n$  lines. Moving up a negative number of lines is the same as moving down.

---

## Motion Commands, *cont'd.*

---

### *Beginning of Line*

**c-A** **Beginning of Line**

Moves the cursor to the beginning of the current line.

With a numeric argument of  $n$ , it moves the cursor to the beginning of the  $n$ th line after the current one, where the current line is numbered 1, the preceding line is numbered 0, and so on.

---

### *End of Line*

**c-E** **End Of Line**

Moves the cursor to the end of the current line.

With a numeric argument of  $n$ , it moves the cursor to the end of the  $n$ th line after the current one, where the current line is numbered 1, the preceding line is numbered 0, and so on.

---

### **Goal Column**

**c-X c-N** **Set Goal Column**

Sets the default column position (*goal column*). The goal column sets point position for **c-N** and **c-P**. It disables the default action of matching the goal column to point's current column and sets the goal column to zero instead. With a numeric argument  $n$ , sets the goal column to  $n$ . **c-U** turns it off (sets back to default state of keeping cursor in same horizontal position for **c-N** and **c-P**).

---

## Motion by Paragraph

---

### Introduction

A paragraph is delimited by:

- A newline followed by blanks (spaces or tabs)
  - A blank line
  - A Page character alone on a line
  - various other mode-dependent things (for example, a line that does not begin with the fill-prefix)
- 

### Forward Paragraph

`m-]`

Forward Paragraph

Moves the cursor forward one paragraph.

Numeric arguments are interpreted as repeat counts; negative numeric arguments reverse the direction of motion.

`m-]` always places the cursor at the end of a paragraph. If the cursor is in the middle of a paragraph, `m-]` moves the cursor to the end of that paragraph.

---

### Backward Paragraph

`m-[`

Backward Paragraph

Moves the cursor one paragraph backward.

Numeric arguments are interpreted as repeat counts; negative numeric arguments reverse the direction of motion.

`m-[` always places the cursor at the beginning of a paragraph. If the cursor is in the middle of a paragraph, `m-[` moves the cursor to the beginning of that paragraph.

---



## Motion by Page

---

### Introduction

Pages are delimited by Page characters. You can insert a Page character by pressing the PAGE key (on an LM-2, press M-CLEAR-SCREEN). The Page delimiter belongs to the page that precedes it and is therefore the last character on that page.

---

### Forward Page

c-X ]

Next Page

Moves the cursor to the beginning of the next page; that is, puts the cursor immediately after the nearest following Page delimiter. If the buffer does not contain a Page delimiter, it goes to the end of the buffer.

With a positive numeric argument  $n$ , it repeats this operation  $n$  times to move forward  $n$  pages. A negative numeric argument  $-n$  moves the cursor backward instead.

c-X [ always places the cursor immediately to the right of the next Page delimiter. If the cursor is immediately to the left of the Page delimiter, c-X ] goes to the beginning of the page after next rather than just moving forward one character.

---

### Backward Page

c-X [

Previous Page

Moves the cursor to the beginning of the previous page; that is, puts the cursor immediately after the nearest preceding Page delimiter. If the buffer does not contain a Page delimiter, it goes to the beginning of the buffer.

With a positive numeric argument  $n$ , it repeats this operation  $n$  times to move backward  $n$  pages. A negative numeric argument  $-n$  moves the cursor forward instead.

c-X [ always places the cursor at the beginning of a page. If the cursor is already at the beginning of the page, c-X [ moves it to the beginning of the previous page.

---

---

## Motion with Respect to the Whole Buffer

---

### Beginning/End of Buffer

`m-<` Goto Beginning

Moves the cursor to the beginning of the buffer.

With a numeric argument  $n$  between 0 and 10, it moves the cursor to a place  $n/10$  of the way (counted in lines) from the beginning of the buffer towards the end.

---

`m->` Goto End

Moves the cursor to the end of the buffer. You can use `m->` if you are in doubt as to the exact place on the screen where the buffer stops.

With a numeric argument  $n$  between 0 and 10, it moves the cursor to a place  $n/10$  of the way (counted in lines) from the end of the buffer towards the beginning.

---

## **Deleting and Transposing Text**

## Deleting vs. Killing

---

### Overview

*Deleting* text merely gets rid of it, but Zmacs deletion commands not only *kill* text but also get it back. These commands save killed text in a *history* stack. Other commands, called *yanking* commands, retrieve elements from the history.

Deletion commands that operate on single characters do not save what they delete. However, by giving them a numeric argument, thus telling them to delete several characters, they too save the deleted text.

The commands that only delete white space do not save it.

---

Zmacs uses several histories:

<i>Type</i>	<i>Description</i>
Kill	History of text deleted or saved. The kill history is shared with the input editor, thus allowing you to move text between files and the Lisp Listener.
Replace	History of arguments to Query Replace (M-X) and related commands. See the chapter "Searching, Replacing, and Sorting", page 84.
Buffer	History of editor buffers visited in this window. See the chapter "Manipulating Buffers and Files", page 100.
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, for example, M-RUBOUT, are not recorded in the history.
Definition	History of names of definitions that have been typed.

History lengths are limitless but the typeout window displays only the first 25 elements of the history. When the history contains more than 25 elements, the screen displays a mouse-sensitive line: *n* more elements in history. Clicking left displays the rest of the history.

Only a single instance of each of these histories exists, shared among all editors, including Zmacs, Zmail, and Dired.

---

---

## Deleting vs. Killing, *cont'd.*

---

### Kill History

The kill history contains deleted text and is the history that saves the results of the commands described in this chapter. It allows you to move text from one editor window to another, for example, from the editor to a Lisp Listener. The *yanking* commands described below retrieve elements from the kill history.

---

### Viewing Kill History

c-@ c-Y

Displays the elements of the kill history (saved text) in a typeout window:

Kill history:

- 1: last piece of killed text
- 2: next-to-last piece of killed text
- 3: this one is a very long piece of killed text...

.  
.  
.

(End of history.)

---

### Viewing the Editor Command History

c-@ c-m-Y

Displays the elements of the editor command history (commands typed) in a typeout window:

Command history:

- 1: Control-X Control-F last-file-read-in
- 2: Help A
- 3: Control-X Control-F other-file-read-in

.  
.  
.

(End of history.)

This command is context-sensitive. When typed at the Lisp listener level, it lists the recent commands typed there. When typed at the minibuffer, it lists the history appropriate to what is being read in the minibuffer, for example, a pathname or the name of a definition.

---

---

**Deleting vs. Killing, cont'd.**

---

**Using the Mouse  
on History Elements**

History elements are mouse-sensitive. Click on an element of the kill history to yank it to point; click on an element of the command history to reexecute it.

---

**Retrieving History Elements**

**c-Y** **Yank**

Yanks back and inserts the last text killed or saved. If you have moved point since you killed the text, put point where you want the killed text to go before pressing c-Y. Point ends up after the text, and mark before the text. An argument of c-U puts point before the text instead. A numeric argument of zero displays the kill history and does not yank anything. A nonzero numeric argument selects an element of the kill history.

---

**c-m-Y** **Repeat Last Minibuffer Command**

Repeats a recent minibuffer command. A numeric argument does the *n*th previous one. An argument of 0 lists the history.

---

**m-Y** **Yank Pop**

Corrects a yank to use a different element of its history. The most recent command must be a yanking command (c-Y, m-Y, or c-m-Y). The retrieved text that was yanked by that command is replaced by the previous element of the relevant history. The history is rotated (that is, the elements remain in the same order, but the pointer to the *current* element moves with each successive m-Y) to bring this element to the top.

A numeric argument of zero displays the history. A positive numeric argument of *n* moves *n* elements back in the history list. A negative numeric argument moves to a newer history element; this only makes sense after you rotate the history.

---

**Kill Merging**

Normally, each kill command pushes a new block onto the kill history. However, two or more kill commands in a row combine their text into a single element on the history, so that a single c-Y command gets it all back as it was before it was killed. This means that you do not have to kill all the text in one command; you can keep killing line after line, or word after word, until you have killed it all, and you can still get it all back at once.

**Deleting vs. Killing, cont'd.**

---

Commands that kill forward from point add onto the end of the previous killed text. Commands that kill backward from point add onto the beginning. This way, any sequence of mixed forward and backward kill commands puts all the killed text into one element without rearrangement.

If a kill command is separated from the last kill command by other commands, it starts a new element on the kill history, unless you tell it not to by saying `c-n-w` (Append Next Kill) in front of it. The `c-n-w` tells the following command, if it is a kill command, to append the text it kills to the last killed text, instead of starting a new element. With `c-n-w`, you can kill several discrete pieces of text and accumulate them to be yanked back in one place.

---

`c-n-w`

Append Next Kill

Makes the next kill command append text to the newest element of the kill history.

---

## Deleting and Transposing Characters

---

### Deleting the Last Character

RUBOUT

Rubout

Deletes the character immediately to the left of the cursor.

If the cursor is at the beginning of a line, RUBOUT deletes the newline character at the end of the previous line, thus appending the current line to the previous one.

With a positive numeric argument of  $n$ , RUBOUT deletes the  $n$  characters immediately to the left of the cursor. With a negative numeric argument of  $-n$ , it deletes the  $n$  characters immediately to the right of the cursor. With any numeric argument, it saves the deleted characters on the kill history.

---

### Deleting the Current Character

c-D

Delete Forward

Deletes the character at the cursor.

If the cursor is at the end of a line, c-D deletes the newline character at the end of the line, thus appending the next line to the current one.

With a positive numeric argument of  $n$ , c-D deletes the  $n$  characters immediately to the right of cursor. With a negative numeric argument of  $-n$ , it deletes the  $n$  characters immediately to the left of cursor. With any numeric argument, it saves the deleted characters on the kill history.

---

### Transposing Characters

c-T

Exchange Characters

Transposes two characters (the ones on each side of the cursor).

If the cursor is not at the end of a line, c-T transposes the character at the cursor and the character to the left of the cursor and advances the cursor one character. The result is that the character to the left of the cursor has been "dragged" one character position to the right. Repeated use of c-T continues to pull that character forward.

This is useful when you are typing and enter two characters in the wrong order (for example, teh for the) — just use c-T to correct the error.

If the cursor is at the end of a line, c-T transposes the two preceding characters.



**Deleting and Transposing Characters, *cont'd.***

---

With a nonzero numeric argument of  $n$ , `c-T` deletes the character to the left of the cursor, moves forward  $n$  characters, and reinserts the deleted character. When  $n$  is negative, the cursor moves backwards.

`c-T` can only be given a numeric argument of zero when the mark is active. In this case, it exchanges the characters at point and mark.

---

---

## Deleting and Transposing Words

---

### Introduction

See the chapter "Moving the Cursor", page 52, for a complete description of how words are delimited.

---

### Deleting the Current Word

`m-D`

Kill Word

Kills the word after the cursor and saves it on the kill history. If the cursor is in the middle of a word, `m-D` kills from the cursor to the end of that word.

With a numeric argument  $n$ , it kills  $n$  words forward from the cursor. If  $n$  is negative, it kills backward.

---

### Deleting the Previous Word

`m-RUBOUT`

Backward Kill Word

Kills the word before the cursor and saves it on the kill history. If the cursor is in the middle of a word, `m-RUBOUT` kills from the cursor to the beginning of that word.

With a numeric argument  $n$ , it kills  $n$  words backward from the cursor. If  $n$  is negative, it kills forward.

---

### Transposing Words

`m-T`

Exchange Words

Transposes the current word and the previous one. If the cursor is at the end of a line, `m-T` transposes the last word on that line and the first one on the next, regardless of the amount or type of white space between them.

With a nonzero numeric argument  $n$ , `m-T` goes to the beginning of the current word, deletes the previous word, goes forward  $n$  words, and reinserts the deleted word. Moving forward a negative amount is equivalent to moving backward. An argument of zero transposes the words at point and mark.

---

## Deleting and Transposing Lisp Expressions

---

### Introduction

See the chapter "Moving the Cursor", page 53, for a complete description of how expressions are delimited.

---

### Deleting the Current Expression

c-m-K

Kill Sexp

Kills the Lisp expression immediately to the right of point and saves it on the kill history.

With a numeric argument of  $n$ , it kills the  $n$  succeeding expressions. It is an error to kill off the end of a containing expression. When the numeric argument is negative, it kills backwards from point the same way.

---

### Deleting the Previous Expression

c-m-RUBOUT

Backward Kill Sexp

Kills the Lisp expression immediately to the left of point and saves it on the kill history.

With a numeric argument of  $n$ , it kills the  $n$  preceding expressions. It is an error to kill off the beginning of a containing expression. When the numeric argument is negative, it kills forward from point the same way.

---

### Deleting the List Containing Current Expression

Kill Backward Up List (c-m-X)

Deletes the list that contains the Lisp expression after point, but leaves that expression itself.

---

---

**Deleting and Transposing Lisp Expressions.** *cont'd.*

---

**Transposing Expressions**

c-m-T

Exchange Sexps

Point must be between two expressions to use this command.

Exchanges the two expressions on either side of point, preserving current indentation.

With a numeric argument of  $n$ , it deletes the expression to the left of point, moves forward  $n$  expressions, and reinserts the deleted expression. With a negative numeric argument, it exchanges expressions in the opposite direction. An argument of zero transposes the expressions at point and mark.

---

---

## Deleting and Transposing Lines

---

### Introduction

See the chapter "Moving the Cursor", page 56, for a complete description of how lines are delimited.

---

### Deleting the Current Line

**c-K** **Kill Line**

Kills a line at a time and saves it on the kill history.

If the cursor is at the end of a line, **c-K** kills the newline, merging the current line with the next one. If the cursor is elsewhere on the line, **c-K** kills the text between the cursor and the end of the current line.

With a numeric argument  $n$ , **c-K** kills up to the  $n$ th newline following the cursor. When  $n$  is negative or zero, **c-K** kills back to the  $1-n$ th newline before the cursor. **c-0 c-K** kills from the cursor back to the beginning of the line that it is on.

---

### Deleting Backward on the Line

**CLEAR-INPUT** **Clear**

Kills backward to the start of the current line and saves it on the kill history. If point is already at the beginning of the line, it kills the previous line. With a numeric argument  $n$ , it kills between point and the start of the  $n$ th line *above* the current line. Use **CLEAR-INPUT** when entering a new line of text, to delete the whole line.

---

### Transposing Lines

**c-X c-T** **Exchange Lines**

Exchanges the current line with the previous one and leaves the cursor at the beginning of the next line.

With a nonzero numeric argument  $n$ , **c-X c-T** deletes the previous line (including the following newline), moves down  $n$  lines, and reinserts the deleted line.

With a numeric argument of zero, **c-X c-T** exchanges the lines at point and mark, advancing both point and mark to the beginning of the next line.

---

## Deleting Sentences

---

### Introduction

See the chapter "Moving the Cursor", page 52, for a complete description of how sentences are delimited.

---

### Deleting the Current Sentence

**m-K** **Kill Sentence**

Kills the text between the cursor and the end of the current sentence, and saves it on the kill history.

With a numeric argument of  $n$ , **m-K** kills the text between the cursor and the end of the  $n$ th sentence after the cursor, *counting* the current sentence. If the argument is negative, **m-K** kills  $-n$  sentences *before* the cursor, counting the current sentence.

---

### Deleting the Previous Sentence

**c-X RUBOUT** **Backward Kill Sentence**

Kills backward one sentence and saves it on the kill history.

With a negative argument, **c-X RUBOUT** kills forward one sentence in a similar manner.

---

## **Working with Regions**

## What is a Region?

---

### Introduction

Many Zmacs commands deal with the region. A region consists of a block of information within the buffer that you want to manipulate as a single entity. You define the area of the region, which can be any size, from characters or chunks of code to pages or the entire buffer.

Zmacs keeps track of one or more locations in a buffer using buffer *pointers*. This section describes:

- The two buffer pointers named *point* and *mark*
  - How Zmacs uses them to define the boundaries of a region
  - The *point-pdl*, a ring of pointers to saved locations
  - *registers*, pointers to locations that you name and save
  - The region-manipulating commands
- 

### Point

Point (shown by the cursor) is the most important buffer pointer. Most editor commands depend on the position of point. Many editor commands, invoked by either the mouse or the keyboard, can be used to position point to the desired location in the buffer. Point points to one end of the region.

---

### Mark

Mark points to the other end of the region. To *mark* a piece of text means to position point and mark on either side of the text, making it the region. The simplest way to mark some text is to position point (using either the mouse or keystrokes) to one boundary (either the beginning or the end) of the text, set the mark there (using the Set Pop Mark command described below), and then reposition point at the other boundary.

Unlike point, the mark can be *active* or *inactive*. When mark is active, the region is shown on the screen by underlining. When mark is inactive, you cannot see it on the screen unless you reactivate it with `c-X c-X`. Although normally you cannot see an inactive mark, Zmacs keeps track of mark when it is inactive and sometimes uses mark in its inactive state. For example, `c-Y` leaves point and mark surrounding what it yanks, but does not activate mark. `c-W` immediately following `c-Y` kills the region even though it is not active. `c-X c-X` after `c-Y` activates mark, making the region visible. However, most commands will not use mark or the region unless it is active.



---

## What is a Region?, cont'd.

---

You can set the mark three ways: when you create a region using the mouse, explicitly with the command Set Pop Mark (`c-SPACE`), or with one of the commands to mark regions (see "Commands to Mark Regions", page 79). When you set the mark, you activate it and make the region appear.

---

### Creating a Region

Create a region using either the mouse or keystrokes — everyone determines their own favorite method.

---

#### *With the Mouse*

The most common way to create a region is with the mouse. Hold down the left mouse button and drag the cursor. Let up the button to mark the end of the region.

Mouse middle creates a region too. It marks the "thing" you point the mouse at, "thing" being mode-dependent (a word or Lisp expression if you point with the mouse at text — a line if you point with the mouse at white space before or after all the text on the line).

---

#### *With Keystrokes*

You can also create a region using keystrokes. After setting the mark, you can move point either forward or backward to define a region in either direction; as you do so, Zmacs highlights the region with underlining.

Typing a self-inserting character or `c-G` deactivates the mark and removes the underlining that highlights the region. The mark does not have an associated cursor like point. When inactive, the mark is invisible, but you can go to it with `c-X c-X`, Swap Point And Mark.

---

### The Point-pdl

Zmacs maintains a special stack of buffer pointers called the *point-pdl*, where *pdl* stands for *push-down list*, another name for a stack.

Zmacs automatically saves point on the *point-pdl* as it executes some commands (for example, `m-<`) that move point great distances. Whenever Zmacs pushes point onto the *point-pdl*, it displays "Point pushed" in the echo area, moves point to its new location, and pushes the previous point down onto the *point-pdl*.

**What is a Region?, cont'd.**


---

By popping the point-pdl, that is, resetting point to its last location as recorded on the point-pdl, Zmacs returns point to where it was when the pdl was last pushed.

---

**Setting/Popping the Mark**

**c-SPACE** **Set Pop Mark**

With no argument, c-SPACE does three things:

1. Puts mark where point is
2. Makes mark active
3. Pushes point onto the point-pdl

Other commands can do each of these operations separately. Creating a region with the mouse sets a mark and makes it active but does not push point.

This command does other things depending on how many c-U's are typed in front of it:

<i>Argument</i>	<i>Action Taken</i>
one c-U	Pops the location on the top of the point-pdl into point (typically puts point where it set the last mark).
two c-U's	Pops the location on the top of the point-pdl and throws it away.

---

**Moving to Previous Points**

**c-m-SPACE** **Move to Previous Point**

Exchanges point and top of point-pdl. With a numeric argument  $n$ , it rotates a ring consisting of point and the top  $n-1$  elements of point-pdl, thus the default argument is 2. With a numeric argument of 1, it rotates the entire point-pdl. A negative numeric argument rotates the ring in the other direction.

---

**c-x c-m-SPACE** **Move to Default Previous Point**

Rotates the point-pdl, the same as c-m-SPACE above except that c-x c-m-SPACE has a default of 3. A numeric argument specifies the number of entries to rotate and sets the new default before rotating the point-pdl.

---

## What is a Region?, *cont'd.*

---

### Showing the Mark

c-x c-x

Swap Point And Mark

Exchanges point and mark. It works even when no region is active. It highlights the text between point and mark.

---

## Registers

---

### Saving and Moving to Locations in Registers

You can assign one-character "names" to locations in the buffer, which can be helpful for setting up a series of places in your text to which you want to return for some reason — to double-check several items without interrupting your text entry or editing, if you are considering a format change that will affect several parallel points, or simply to return quickly and easily to rough spots that require further work.

---

**c-X S** Save Position

Saves the current location in a register. It prompts for a one-character register name.

---

**c-X J** Jump to Saved Position

Moves point to a position that was saved in a register. It prompts for a register name and switches buffers to move to the saved position, if necessary.

---

### Saving and Inserting Regions in Registers

**c-X X** Put Register

Copies the text of the region into a register. It prompts for a register name. With a numeric argument, it deletes the region from the buffer after copying it.

---

**c-X G** Open Get Register

Inserts text from a specified register into the buffer. It prompts for the name of the register. It overwrites blank lines in the buffer the way RETURN does (using the command Insert Crs). It leaves the mark before the inserted text and point after it. With a numeric argument, it puts point before the text and the mark after.

---

**List Registers (m-X)**

Displays names and contents of all defined registers. It shows the name of the register and whether it contains a position or text. If the register contains a position, it tells which character on the line

## Registers, cont'd.

---

the position is at, and shows the first 50 characters on that line. If the register contains text, it shows the first 50 characters on the first line of that text.

List of all registers:

D (text)        This text was marked as a region and saved here  
1 (position)    Char 0. in "another line containing a position"  
Done.

---

### View Register (m-X)

Displays the contents of a register in the typeout window. It prompts for a register name and then tells whether the register contains a position or text:

Register A contains a position: Character 0 in this line:  
this is the line  
or  
Register A contains text:

---

### Kill Register (m-X)

Kills a register.

---

## Commands to Mark Regions

---

### Overview

To *mark* a piece of text means activating mark and then positioning point and mark on either side of the text, making it the region. The simplest way to mark some text is to go to one end of the text, set the mark there (using the Set Pop Mark command described earlier in this section), and go to the other end of the text. However, there are several convenient commands for marking different amounts of text, which are described below.

---

### By Words

**m-@** Mark Word

Puts the mark at the end of the current word. With a numeric argument of  $n$ , m-@ puts the mark  $n$  words forward from point.

---

### By Lisp Expressions

**c-m-@** Mark Sexp

Marks the current expression by putting mark at the end. With a numeric argument  $n$ , it moves forward  $n$  expressions and puts the mark there. See c-m-F for a more detailed description of how to move forward  $n$  expressions.

---

**c-m-H** Mark Definition

Puts point and mark around the current definition.

---

### By Paragraphs

**m-H** Mark Paragraph

Puts the mark at the end of the current paragraph and moves point to the beginning, so that the current paragraph becomes the region. With a numeric argument  $n$ , m-H puts point at the beginning of the current paragraph and marks  $n$  paragraphs forward from there.

---

### Example

m-3H marks the current paragraph and the following two; m- -1H marks the preceding paragraph. When marking preceding paragraphs, point is left at the end of the region, and when marking current and succeeding paragraphs, point is left at the beginning of the region.

---

---

## Commands to Mark Regions

---

### By Pages

c-X c-P Mark Page

Puts the mark at the end of the current page and moves point to the beginning, so that the current page becomes the region.

With a numeric argument of  $n$ , c-X c-P marks the  $n$ th page after the current one. If  $n$  is zero, this is the current page; if  $n$  is negative, this page comes *before* the current page.

---

### By Buffers

c-X H Mark Whole

Marks the whole buffer by putting point at the beginning and the mark at the end. With any numeric argument, c-X H puts the mark at the beginning and point at the end.

---

### *From Here to End of Buffer*

c-> Mark End

Marks from the cursor to the end of the buffer by putting the mark at the end of the buffer.

---

### *From Here to Beginning of Buffer*

c-< Mark Beginning

Marks from the cursor to the beginning of the buffer by putting the mark at the beginning of the buffer.

---

---

## Region-Manipulating Commands

---

### Saving the Region

m-W Save Region

Puts region on kill history list without deleting it. (See also the section "Kill Merging", page 64, including the description of the Append Next Kill command, c-m-W.)

---

### Deleting the Region

c-W Kill Region

Deletes the region. If there is no region, c-W produces an error. This command ignores numeric arguments and places the deleted text on the kill history list. (See also the section "Retrieving History Elements", page 64, including the description of the Yank command, c-Y.)

---

### Compiling the Region

c-sh-C Compile Region

Compile Region (m-X)

Compiles the region, or if no region is defined, the current definition.

---

### Transposing Regions

c-X T Exchange Regions

Exchanges two regions delimited by point and last three marks. After transposing regions, you can undo the effect of this command by invoking it again.

---

### Hardcopying the Region

Hardcopy Region (m-X)

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

---



## Region-Manipulating Commands, *cont'd.*

---

### Filling the Region

When Zmacs *fills* text it breaks it up so that it does not extend past the *fill column*. The fill column determines the right margin, and is the first column in which text is not to be placed by `m-Q`, `m-G`, or Auto Fill Mode formatting. In addition, the *fill prefix*, if set, is inserted:

- at the beginning of each new line typed in while in Auto Fill Mode
- at the beginning of each line in a paragraph for `m-Q` and each line in a region for `m-G`

The fill prefix determines the left margin, and is empty unless set to contain some combination of spaces and characters. If you do not set the fill prefix, the left margin is the left edge of your Zmacs window. For example, to insert five spaces at the beginning of every line, insert them at the beginning of the current line, and with point at column six, use `c-X ..`. To turn this fill prefix off, put point at the beginning of a line, and use `c-X .` again.

*Adjusting* or *justifying* text inserts extra spaces between the words to make the right margin come out exactly even.

---

<code>m-Q</code>	Fill Paragraph
------------------	----------------

Fills the current (or next) paragraph. A positive argument means to adjust rather than fill.

---

<code>m-G</code>	Fill Region
------------------	-------------

Fills the current region. A positive argument means to adjust rather than fill.

---

<code>c-X .</code>	Set Fill Prefix
--------------------	-----------------

Defines Fill Prefix from the current line. All of the current line up to point becomes the Fill Prefix. Fill Region starts each nonblank line with the prefix (which is ignored for filling purposes). To stop using a Fill Prefix, do a Set Fill Prefix at the beginning of a line.

---

---

## Region-Manipulating Commands

---

### Other Region- related Commands

<i>Name and Invocation</i>	<i>See Page</i>
Uppercase Region c-X c-U	148
Lowercase Region c-X c-L	148
Uppercase Code in Region (m-X)	148
Lowercase Code in Region (m-X)	148

---

## **Searching, Replacing, and Sorting**

## Searching

---

### Overview

Like other editors, Zmacs has commands for searching for an occurrence of a string. Zmacs search commands are *incremental*; that is, they begin to search as soon as you type the first character.

This section describes how to search incrementally forward and backward in the buffer, as well as a method for specifying a complete search string first and then specifying a direction in which to search.

---

### Incremental Search

The command to search is `c-S` (Incremental Search). `c-S` reads in characters and positions the cursor at the first occurrence of the characters that you have typed. If you type `c-S` and then `t`, the cursor moves right after the first `t`. Type an `r`, and see the cursor move to after the first `tr`. Add a `y` and the cursor moves to the first `try` after the place where you started the search. At the same time, the `try` has echoed at the bottom of the screen. Stop typing when you have typed enough characters to identify the place you want.

If you type a mistaken character, you can rub it out. After the `try`, typing a RUBOUT makes the `y` disappear from the bottom of the screen, leaving only `tr`. The cursor moves back to the `tr`. Rubbing out the `r` and `t` moves the cursor back to where you started the search. To exit from the search, press END or ESCAPE (ALTMODE does the same thing on an LM-2). You can also use ABORT to exit from the search. To abort out of the search and return to the original starting point in the buffer, use `c-G`.

If you want to search for something else, press CLEAR-INPUT to get rid of the current search string. You're still in the search loop, so type another search string.

If the string cannot be found with `c-S`, type `c-R` to search backward for the default string. Zmacs remembers the default search string — you can reinvoke the search at any time using `c-S c-S`, to search forward for it, or `c-R c-R` to search backward.

`c-S`

Incremental Search

Searches for a character string while you type it, searching forward to the end of the buffer. It prompts for a string in the echo area with `I-Search:.` As you type characters in, `c-S` displays the accumulating string in the echo area and searches for it at the same time. If no string is found, it displays `Failing I-Search:.` When it locates the string, it puts the cursor after it so that repeated `c-Ss` locate subsequent occurrences of the default string in the buffer.

## Searching, *cont'd.*

---

RUBOUT	Removes a character and backs up the search to the last match.
ESCAPE	When typed before any search characters, switches to String Search (see page 88).
END	Exits the search (ESCAPE also works for the 3600, ALTMODE works for the LM-2).
c-G	Exits the search and returns to original starting point in the buffer.
c-Q	Quotes the next character, to prevent it from terminating the search.
c-S	Repeats the search.
c-R	Reverses the search to search backwards.

If c-S or c-R is the first character typed, the previous search string is used again as the default. Entering any other command character terminates the search (and then executes that command).

---

## Reverse Incremental Search

c-R, Reverse Incremental Search, works exactly the same way as c-S, except that it searches *backward* towards the top of the buffer from point, instead of forward.

c-R Reverse Incremental Search

Searches for a character string while you type it, searching backward to the beginning of the buffer. It prompts for a string in the echo area with Reverse I-Search:. As you type characters in, c-R displays the accumulating string in the echo area and searches for it at the same time. If no string is found, it displays *Failing Reverse I-Search:*. When it locates the string, it puts the cursor in front of it so that repeated c-Rs locate previous occurrences of the default string in the buffer.

RUBOUT	Removes a character and backs up the search to the last match.
ESCAPE	When typed before any search characters, switches to Reverse String Search (see page 88).
END	Exits the search (ESCAPE also works for the 3600, ALTMODE works for the LM-2).
c-G	Exits the search and returns to original starting point in the buffer.

**Searching.** *cont'd.*


---

c-Q	Quotes the next character, to prevent it from terminating the search.
c-S	Reverses the search to search forward.
c-R	Repeats the search.

If c-S or c-R is the first character typed, the previous search string is used again as the default. Entering any other command character terminates the search (and then executes that command).

---

**String Search**

The string search command, invoked by c-S ESCAPE (c-S ALTMODE on an LM-2), lets you type in the entire string and specify the direction in which to search before starting the search.

c-S ESCAPE String Search

Searches for a specified string, according to the arguments given with the special characters below. Another c-S always begins the search. It prompts in the echo area String Search:. It saves previous string search commands on a ring, retrievable with c-D. The ring contains three elements and can be rotated with repeated c-Ds. While you are entering the search string, the following characters have special meanings:

c-B	Searches forward from the beginning of the buffer.
c-E	Searches backwards from the end of the buffer.
c-F	Leaves point at the top of the window, if the window must be recentered.
c-G	Aborts the search.
c-D	Gets a string to search for from the ring of previous search strings.
c-L	Redisplays the typein line.
c-Q	Quotes the next character.
c-R	Reverses the direction of the search.
c-S	Does the search, then comes back to the search command loop.
c-U	Erases all characters typed so far (CLEAR-INPUT also works for the 3600).
c-V	Delimited Search: Searches for occurrences of the string surrounded by delimiters.

**Searching**, *cont'd.*

---

<code>c-W</code>	Word Search: Searches for words in this sequence regardless of intervening punctuation, whitespace, newlines, and other delimiters.
<code>c-Y</code>	Appends the string on top of the string ring to the search string.
<code>RUBOUT</code>	Rubs out the previous character typed.
<code>END</code>	Does the search and exits (ESCAPE also works on a 3600; ALTMODE on an LM-2).

If you search for an empty string, it uses the default. Otherwise, the string you type becomes the default, and the default is saved unless it is a single character.

---

## Locating and Replacing Strings Automatically

---

### Overview

`c-z`, `Replace String`, searches forward for a string and replaces that string with another. `c-z` prompts for the string to be replaced, reads the string from the minibuffer, and then reads the replacement string. After it goes through the buffer trying to make the replacements, it tells you how many replacements it made (1. replacement.), or that it made none.

You can also substitute one string for another *selectively* throughout the buffer, with `m-z`, `Query Replace`. `m-z` prompts first for the string to be replaced (`Query-replace some occurrences of:`), and then for the string to replace it with (`Query-replace some occurrences of "string" with:`). Terminate each string you specify with `RETURN`. `m-z` locates each occurrence and lets you decide what to do about each one.

---

### Making Global Replacements

`c-z` `Replace String`  
`Replace String (m-x)`

Replaces all occurrences of a given string with another, where the string can be characters, words, or phrases. It prompts first for the string to remove and second for the string to replace it with. A numeric argument *n* means to make *n* replacements. By default, it begins at point and replaces all occurrences of the first string that occur *after* point in the buffer. Usually it attempts to match the case of the replacements with the case of the string being replaced. This behavior is controlled by the Zmacs variable `Case Replace P` (default `t`). When it is null, case matching does not take place. (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193).

---

### Querying While Making Global Replacements

`m-z` `Query Replace`  
`Query Replace (m-x)`

Starting at point, replaces a string through the rest of the buffer, asking about each occurrence, where the string can be characters, words, or phrases. It prompts for each string. You first give it `STRING1`, then `STRING2`, and it finds the first `STRING1`, displaying it in context. You respond with one of the following characters:

<code>SPACE</code>	Replaces it with <code>STRING2</code> and shows next <code>STRING1</code>
<code>RUBOUT</code>	Leaves this <code>STRING1</code> , but shows next <code>STRING1</code>



---

## Locating and Replacing Strings Automatically. *cont'd.*

---

,	Replaces this STRING1 and shows result, waiting for a SPACE, c-R, or ESCAPE
Period	Replaces this STRING1 and ends query replace
c-G	Leaves this occurrence of STRING1 unchanged and terminates the query replace
ESCAPE	Same as c-G
^	Returns to site of previous STRING1
c-W	Kills this STRING1 and enters recursive edit
c-R	Enters editing mode recursively. Press END to return to Query Replace.
c-L	Redisplays screen
!	Replaces all remaining STRING1s without asking

Entering any other character terminates the command. Usually the command attempts to match the case of the replacements with the case of the string being replaced. This behavior is controlled by the Zmacs variable Case Replace P (default **t**). When it is null, case matching does not take place. (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193).

If you give a numeric argument, it does not consider STRING1s that are not bounded on both sides by delimiter characters.

---

### Querying While Making Multiple Global Replacements

While doing multiple query replacements, you can specify the replacement strings either from the minibuffer or from another buffer altogether.

---

### *Replacements from the Minibuffer*

#### Multiple Query Replace (m-X)

Performs query replace (see the description for Query Replace (m-X), page 90) using many pairs of strings at the same time, where the strings can be characters, words, or phrases. Strings are read in alternate minibuffers; when you finish entering all strings, press RETURN twice. An argument means that the strings must be surrounded by delimiter characters. A negative argument means that the strings must be delimited atoms, rather than words.

---

---

**Locating and Replacing Strings Automatically.** *cont'd.*

---

**Replacements from  
Another Buffer****Multiple Query Replace From Buffer (m-X)**

Performs query replace (see the description for Query Replace (m-X), page 90) using many pairs of strings *supplied from the specified buffer*. The current buffer should contain a STRING1, a space, and a STRING2. Put quotation marks around any string that contains a space, tab, backspace, semicolon, or newline character. Lines in the buffer that begin with a semicolon or are blank are ignored. In other words, each string in the buffer is a Lisp string, but quotation marks can be omitted if the string contains no special characters.

---

**Other Types of Replacements**

Besides making string replacements in text, Zmacs commands replace:

- A region into the kill history
  - Evaluated code into the buffer
  - The value of LET into its variable
  - A string for delimited atoms
- 

**Query Replace Last Kill****Query Replace Last Kill (m-X)**

Replaces the first item in the kill history with the region.

---

**Evaluate and  
Replace Into Buffer****Evaluate and Replace Into Buffer (m-X)**

Evaluates the next Lisp expression following point and replaces it with the printed representation of its value.

---

**Query Replace LET Binding****Query Replace Let Binding (m-X)**

Replaces variable of LET with its value. Point must be after or within the binding to be modified.

---

**Locating and Replacing Strings Automatically**, *cont'd.*

---

*Atom Query Replace***Atom Query Replace (m-x)**

Performs query replace (see the description for Query Replace (m-x), page 90) for delimited atoms.

---

## Tags Tables and Search Domains

---

### Introduction

*Tags tables*, a means of global searching and replacing, allow you to make sweeping changes to groups of files without having to explicitly locate each file. *Tags tables* are sets of buffers and files. *Tags files* provide a list of the names of files that belong together as part of a system and a list of names and locations of definitions within the files. The file names are made into a tags table; the definition names are added to the completion table.

You could use tags tables, for example, to:

- Search for all references to a certain variable and alter them consistently
  - Search for all occurrences of an obsolete term and update it
  - Search for all functions that send a certain message
- 

### How They Work

First, you specify the buffers or files that will make up the tags table (see "Specifying and Listing Tags Tables" below). Then you can perform an operation (see "Performing Operations with Tags Tables" below). Zmacs performs the operation on the files within the tags table that you have specified.

---

### Example

Suppose you want to perform a tags query replace in several files. Use Tags Query Replace (m-x) (described in detail below) to begin. The minibuffer prompts as in Query Replace (m-x) for the string to be replaced and the replacement string. The operation begins and Zmacs displays Control-. is now Continue query replacement of "string-old" with "string-new"; as it displays each occurrence, you deal with each one using the appropriate response characters. Tags Query Replace goes through all the files specified in the tags table, listing their names in the minibuffer and stopping at each occurrence of "string-old". When it finishes searching all the files, it displays No more files.

---

### Specifying and Listing Tags Tables

Select All Buffers As Tag Table (m-x)

Selects all buffers currently read in. It creates a support buffer (see "Support Buffers" below) called \*Tag-Table-N\*, which contains a list of the names of all the buffers.

---

## Tags Tables and Search Domains, *cont'd.*

---

### Select Tag Table (m-X)

Makes a tags table current for commands like tags search. It prompts in the minibuffer for the name of the tags table to use.

---

### Select System As Tag Table (m-X)

Creates a tags table for all files in a system defined by **defsystem**. It prompts in the minibuffer for the name of a system — press **HELP** to see a display of system names. It selects the system but does not read the files in.

---

### List Tag Tables (m-X)

Lists in the typeout window the names of all the tags tables, and for each one shows the files it contains.

---

## Performing Operations With Tags Tables

### Tags Search (m-X)

Searches for the specified string within files of the tags table. It prompts in the minibuffer for the search string. If there is no current tags table, it prompts for one.

Zmacs displays in the echo area the name of each of the files in the tags table as it searches each file for the specified string. As Zmacs begins the operation and finds the first occurrence, it displays **Point pushed.** in the minibuffer and moves the cursor to the occurrence. After you deal with that occurrence, use **c-.**, the Edit Definition command (described below), to tell the command to locate the next occurrence. Go through the specified files using **c-.** to the end.

---

### Tags Query Replace (m-X)

Replaces occurrences of one string with another within the files of the tags table, asking about each occurrence. It prompts first for the string to remove and second for the string to replace it with. You first give it **STRING1**, then **STRING2**, and it finds the first **STRING1**, displaying it in context. You respond with one of the following characters:

**SPACE**                      Replaces it with **STRING2** and shows next **STRING1**

**Tags Tables and Search Domains, cont'd.**


---

RUBOUT	Does not replace this occurrence, but shows next STRING1
,	Replaces this STRING1 and shows result, waiting for a SPACE, c-R, or ESCAPE
Period	Replaces this STRING1 and terminates the query replace
c-G	Leaves this occurrence of STRING1 unchanged and terminates the query replace
ESCAPE	Same as c-G
^	Returns to site of previous STRING1 (actually, pops the point-pdl)
c-W	Kills this STRING1 and enters recursive edit
c-R	Enters editing mode recursively. Press END to return to Query Replace.
c-L	Redisplays screen
!	Replaces all remaining STRING1s without asking

Entering any other command character terminates the command. Usually the command attempts to match the case of the replacements with the case of the string being replaced. This behavior is controlled by the Zmacs variable Case Replace P (default t). When it is null, case matching does not take place. (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193).

If you give a numeric argument, it does not consider STRING1s that are not bounded on both sides by delimiter characters.

---

**Tags Multiple Query Replace (m-X)**

Performs tags query replace (see the description for Tags Query Replace above) using many pairs of strings at the same time, where the strings can be characters, words, or phrases. Strings are read in alternate minibuffers; when you finish entering all strings, press RETURN twice. An argument means that the strings must be surrounded by delimiter characters. A negative argument means that the strings must be delimited atoms, rather than words.

---

**Tags Multiple Query Replace From Buffer (m-X)**

Replaces occurrences of any number of strings with other strings within the tags table files, asking about each change. The current

---

## Tags Tables and Search Domains, *cont'd.*

---

buffer should contain a STRING1, a space, and a STRING2. Put quotation marks around any string that contains a space, tab, backspace, semicolon, or newline character. Lines in the buffer that begin with a semicolon or are blank are ignored. In other words, each string in the buffer is a Lisp string, but quotation marks can be omitted if the string contains no special characters.

A positive numeric argument means to consider only the cases where the strings to replace occur as a word (rather than within a word). A negative numeric argument means to consider only delimited atoms, rather than words.

This command has the same options as Tags Query Replace (see above).

---

### Find Files in Tag Table (m-X)

Reads every file in the selected tags table into the editor. If there is no current tags table, it prompts for the name of one, which you can specify as a file (F), all editor buffers (B), or a system (S).

---

### Visit Tag Table (m-X)

Creates a tags table by reading in a tags file. First, it reads in the specified tags file. It prompts for a file name from the minibuffer. Next, it goes through the tags file and marks the name of each tag as being a possible section of its file. The Edit Definition command (m-.) uses these marks to figure out which file to use.

It uses a support buffer (see "Support Buffers" below) to hold the elements of the tags table and another support buffer to hold the state of a pending operation involving all the files in the tags table. Each contains the names of the files.

---

## Support Buffers

Zmacs creates *support buffers* to save lists that it creates as part of the execution of some commands:

- Tags table commands.
- Edit Buffers (m-X).
- View File (m-X).
- Lists for Edit Definition (m-.), when more than one definition exists.
- Buffers for Dired (m-X).
- Everything that edits a sequence of definitions, as in List Callers (m-X) or List Methods (m-X).

This means that you can examine the buffers containing the lists even after you have done some editing.

## Tags Tables and Search Domains, *cont'd.*

---

`c-X c-B`, the List Buffers command, displays these support buffers in the listing of buffers. Their names are, for example, `*Definitions-1*`, `*Tags-Search-1*`, and `*Tags-Query-Replace-1*`.

To avoid proliferation of editor buffers, Zmacs reuses support buffers in most cases, so that it usually saves no more than two of each type of support buffer at a time.

---

### *Possibility Buffers*

Each time you use a command that generates a set of possibilities (for example, Tags Search (`m-X`) and Tags Query Replace (`m-X`)), it creates a possibility buffer for that set and pushes the set of possibilities onto a stack. `c-.`, Next Possibility, extracts the next item from the set at the top of the stack. The set is popped from the stack when no more items remain in it. Several informational messages are associated with this facility. When the whole possibilities stack is empty and you have nothing more pending it displays:

No more sets of possibilities.

---

### *Displaying the Next Possibility*

`c-.` Next Possibility

Selects the next possibility for the current set of possibilities. With a negative argument, pops off a set of possibilities. An argument of `c-U` or any positive number displays the remaining possibilities in the current set. With an argument of zero, selects the current buffer of possibilities.

See the chapter "Editing Lisp Programs", page 153, for a description of the Edit Definition and Edit Callers commands.

---

### *Example*

Suppose you had been using `c-.` to move through the set provided by Tags Search and you then used Tags Query Replace to push a new set of possibilities onto the stack. When you finished the set provided by Tags Query Replace, you would see a message like the following to notify you that the empty set had been popped off the stack and the set of possibilities for Tags Search had been reinstated.:

`c-.` is now Search for next occurrence of "string"

The position of point in the support buffer indicates the next item for Next Possibility (`c-.`). You can select the support buffer and move point manually in order to skip or redo possibilities.



**Tags Tables and Search Domains, *cont'd.***

---

Typing `c-` while in a support buffer that is not at the top of the possibilities stack moves it to the top, prints an appropriate message, then takes the next possibility from that support buffer.

---

---

## Sorting

---

### Overview

The following commands alphabetically sort a region by line, paragraph, or whatever *sort key* you specify.

---

#### Sort Lines (m-X)

Sorts the region alphabetically by lines.

---

#### Sort Paragraphs (m-X)

Sorts the region alphabetically by paragraphs.

---

#### Sort Via Keyboard Macros (m-X)

Sorts the region, prompting for actions to define the *records* (the units of the region to be rearranged) and the *sort keys* (the fields in the records that are compared alphabetically to determine the new order of records). It prompts you to define the records and sort keys by performing positioning commands. It prompts for three actions:

1. Move to the beginning of the sort key (that is, move the cursor to the beginning of the field upon which to sort).
2. Move to the end of the sort key (that is, move to the end of the sort field).
3. Move to the end of the sort record (that is, move to the end of the record containing that field).

For each, it records the keystrokes that you use (as keyboard macros) and plays those back to find and sort the records in the region.

---

## **Manipulating Buffers and Files**

---

## Working With Buffers and Files

---

### Overview

*Files* are semipermanent collections of information stored safely outside the Zmacs environment. *Buffers*, on the other hand, are more dynamic, temporary collections of information, used by Zmacs for manipulating text. Buffers live in the active Zmacs environment. Each buffer has its own point and mark as well as other associated information.

We say we use Zmacs to "edit files", but what we really do is copy a file into a buffer created for the purpose, edit the buffer, and then write out a new version of the file from the edited buffer. The old version of the file is retained, to be deleted explicitly when appropriate. Successive versions of files are distinguished by *version number*, a component of the file name that is incremented with each new revised copy (except on file server hosts like UNIX that do not have version numbers).

Zmacs allows multiple buffers, so that you can edit many files simultaneously. Usually only one buffer is visible on the screen at a time. You can, however, divide the screen into multiple windows so that you can view the contents of several buffers at once.

Zmacs keeps track of the association between files and buffers. If you are editing a file's contents in a buffer, Zmacs gives that buffer the same name as that of the file being edited.

---

### Buffer and File Names

Both buffers and files have long names that indicate the host directory as well as the file name (and version, where supported). Hence completion is a necessary aid and is always provided for entering buffer and file names.

---

## Working With Buffers and Files. *cont'd.*

---

### Buffer Flags for Existing Files

Each buffer has a *modification flag* that tells whether the buffer has been changed to be different from the associated file. You can see the modification flag by clicking on either the List Buffers command or the Kill or Save Buffers command in the editor menu (editor menu is click right once), or by pressing `c-X c-B` for List Buffers. The modification flag is cleared when:

- The file is read into the buffer from the file system.
- The buffer is *saved*, that is, whenever its contents are written out to the associated file. As soon as its contents are modified thereafter, the modification flag is set and Zmacs displays an asterisk (\*): (1) in the mode line to the right of the buffer name, and (2) whenever it displays output from the List Buffers command.

---

### Buffer Flags for New Files

The List Buffers (`c-X c-B`) command uses the plus sign (+) to mark new files that have not been saved. In addition, it uses + to mark new buffers, not associated with files, that have text in them. This helps when you put text into a new buffer and later want to be reminded to write that buffer to a file.

---

---

## Selecting, Listing, and Examining Buffers

---

### Current Buffer

At all times when using Zmacs, you have one *selected* buffer, which is the buffer that you are actively editing. This is the buffer whose cursor moves when you type `c-F` and in which all other current activity takes place until you switch buffers.

---

### Buffer History

With a single Zmacs window on the screen, the editor keeps one buffer history, the *global history list*, which remembers the previous-buffer history (stack history) of that window. The top buffer in the stack is the currently selected one. Usually, when a buffer is selected, it is dredged out of the stack and put on top. The buffers near the top are usually the most recently used. Each time you change buffers Zmacs offers the name of the most recently used buffer as the default buffer name.

When we refer to the *n*th buffer, we mean the *n*th buffer in Zmacs's stack of buffers.

Every additional window maintains its own buffer history, but the global history list continues to display an entry for every buffer in every window.

When you create a new window, Zmacs initially takes the history list for the new window from the global history list. From then on, as you switch from buffer to buffer within that window, the list for that window reflects the history of those changes in chronological order. This affects particularly `c-M-L` (Select Previous Buffer) and the default for `c-X B` (Select Buffer).

The global history list still exists and is used for name completion and `c-X c-B` (List Buffers).

---

---

## Buffer Commands

---

### Changing Buffers

**c-X B** Select Buffer

Prompts for the name of a buffer and selects that buffer, displaying its contents on the screen. If you press END or RETURN instead of a name, it reselects the second most recently selected buffer.

Using completion, it takes the string you enter and tries to complete it to an existing buffer name:

- When completion is successful, it selects that buffer.
- When completion is unsuccessful, (there is no buffer with the name given), it either waits for you to type more characters (if there are multiple possible completions) or it beeps to give you a chance to correct a typing error (if there is no possible completion). A subsequent response of c-RETURN creates a new buffer with the specified name and selects it.

If you precede the c-X B command with a numeric argument, Zmacs prompts for the name of the buffer and then creates and selects it.

---

**c-m-L** Select Previous Buffer

Selects a previously selected buffer. With a numeric argument *n*, it selects the *n*th previous buffer. The default argument is 2. When the argument is 1, it rotates the entire buffer history. A negative argument means to rotate the other way. An argument of zero displays the buffer history, which is mouse-sensitive.

---

**c-X c-m-L** Select Default Previous Buffer

With a numeric argument *n*, this is exactly the same as c-m-L. Without a numeric argument, this command *remembers the last numeric argument it received* and uses that as its argument this time.

This is useful if you happen to be working with the top few buffers on the buffer stack and want to cycle among them without having to remember how many there are.

---

**Buffer Commands**, *cont'd.***Listing Buffers****c-X c-B****List Buffers**

Lists all the currently existing buffers in the typeout window, along with the editor mode of the buffer and the name of the associated file, if any. For buffers with associated files, it displays the version number of the file, if any. If there is no associated file, **c-X c-B** gives the size of the buffer in lines instead. For Dired buffers, it displays the pathname used for creating the buffer. It lists modified buffers with an asterisk. It lists the buffers sorted in stack order. You can inhibit this sorting by setting the global variable **zwei:\*sort-zmacs-buffer-list\*** to **nil** (default is **t**).

With an argument of **c-U**, it prompts for a substring and then lists all buffers whose names contain that substring.

The buffer names are mouse sensitive. Click right on the name of the buffer for a menu of operations (Kill, Not Modified, Save, Select) for that buffer. You can select one of the buffers by clicking left on its name.

*Example*

Buffers in Zmacs:

Buffer name:	File Version:	Major mode:
+ file1 /dess/zmacs VIXEN:		(Fundamental)
= *Dired-1*	VIXEN: /dess/zmacs/*	(Dired)
* doc.mss /dess/zmacs VIXEN:		(Text)
*Buffer-1*	[1 line]	(Fundamental)

+ means new file or non-empty non-file buffer. \* means modified file.  
= means read-only.

**Editing Buffers**

Edit Buffers (**c-m-X**) is not part of the standard comtab. It is similar to List Buffers (**c-X c-B**), except that the buffer listing that Edit Buffers produces is a buffer in its own right. (See "Setting Editor Variables in Init Files", page 196, for an example showing how to make **c-X c-B** call Edit Buffers instead of List Buffers.) It contains one line for each of the buffers in the editor.



---

## Buffer Commands, *cont'd.*

---

### Edit Buffers (c-m-X)

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. The buffer is read-only; like the Directory editor (Dired) buffer, you can move around in it by searching and with commands like c-N and c-P.

The lines in the list are not mouse-sensitive. With the cursor on the line for a buffer, the following single character commands apply to that buffer:

RUBOUT	Undeletes buffer above the cursor.
SPACE	Selects the specified buffer immediately.
D	Marks the buffer for deletion (K, c-D, c-K are synonyms).
U	Undeletes either the buffer on the current line or the buffer on the line above.
S	Marks the buffer for saving.
~	Marks the buffer for setting not modified.
X	Executes an extended command (same as m-X).

---

### Viewing a Buffer

View Buffer is for when you want to just look at a buffer, not edit it.

c-X V View Buffer  
View Buffer (m-X)

Prompts for the name of a buffer and prints out the buffer contents for viewing only in the typeout window. If there is more than a screenful, it pauses between screenfuls, displaying a --MORE-- message at the bottom.

SPACE	Displays the next screenful.
BACKSPACE	Displays the previous screenful.
RUBOUT	Exits.

Anything else exits and is executed as a command.

---

---

**Buffer Commands, cont'd.**

---

**Hardcopying the Buffer****Hardcopy Buffer (m-X)**

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

---

**Renaming the Buffer****Rename Buffer (m-X)**

Prompts for a new name for the current buffer and changes the name accordingly. This operation removes any file association that the buffer had.

---

**Writing Out All Buffers****Save All Files (m-X)**

Offers to write out each buffer that is associated with a file. It prompts in the typeout window with the name of each buffer: Save file old.lisp /dass/pubs/pgs VIXEN:? (Y or N).

---

**Encrypt Buffer (m-X)**

Encrypts the contents of the buffer. It prompts for a key and does not echo it as you type it. It prompts for the same key again, just in case you mistyped it because of the lack of echoing, and makes sure you typed it the same both times. The encryption algorithm is the same one used by the Hermes mail-reading system.

---

**Decrypt Buffer (m-X)**

Decrypts the contents of an encrypted buffer. It prompts for a key and does not echo it as you type it. The encryption key given for decrypting must match the one used for encrypting. The encryption algorithm is the same one used by the Hermes mail-reading system.

---

**Reading a File  
Into a New Buffer****c-X c-F****Find File**

Prompts for the name of a file and looks for a buffer currently associated with that file. If one is found, c-X c-F selects it. Otherwise, it creates a new buffer and reads that file into it.

---

---

## Buffer Commands, *cont'd.*

---

### Reading a File Into an Existing Buffer

The `c-X c-V` command, Visit File, is primarily useful when you type in a mistaken file name after `c-X c-F` and Zmacs responds (New File). You can simultaneously read in the correct file and get rid of the unwanted buffer with Visit File.

`c-X c-V` Visit File

Prompts for the name of a file and reads that file into *the current buffer*. This action associates the current buffer with the specified file. This command can only be used if the current buffer is not already associated with an existing file.

---

### Writing the Buffer Contents to a File

`c-X c-W` Write File

Prompts for the name of a file and writes out the contents of the current buffer to the specified file. This changes the current buffer's name and associates it with the specified file. Subsequent saves using `c-X c-S` save to the newly specified file. This operation clears the modification flag.

---

### Saving the Buffer Contents to the File

`c-X c-S` Save File

Writes the contents of the current buffer out to the associated file and clears the modification flag. It does not write the file if the buffer is unchanged from when the file was last visited or saved. It reads a file name from the minibuffer if the current buffer does not have an associated file.

---

### Re-reading a File Into the Buffer

Revert Buffer (`m-X`)

Reads a file into the buffer that it is associated with. It prompts for a buffer name, defaulting to the current buffer. The prompt serves as a confirmation, since Revert Buffer (`m-X`) throws away any modifications made to the buffer since you last saved or read the file. This command is useful if you have damaged the buffer and want to start over or if the associated file is more current than the buffer. This operation clears the modification flag.

---

---

**Buffer Commands, cont'd.**

---

**Creating a  
Fundamental  
Mode Buffer****Find File In Fundamental Mode (m-X)**

Creates a fundamental mode buffer containing the file. This is useful because Zmacs does not parse the file while reading it in, thus the names of the functions in the file do not conflict with those already known to completion in `m-` and similar commands. This command is necessary if the normal parsing of a Lisp Mode file signals an error, preventing it from being read into the editor to correct the cause of the error.

---

**Associating a File  
With a Buffer****Set Visited File Name (m-X)**

Prompts for the name of a file and associates the current buffer with that file. This command does *not* read the specified file into the buffer. Effectively, the current contents of the buffer are declared to be the new intended contents of the specified file. This command should be used with caution to avoid unintentionally destroying the old contents of the specified file.

---

**Destroying Buffers****c-X K****Kill Buffer**

Prompts for the name of a buffer and destroys that buffer. If you press END or RETURN instead of a name, `c-K` destroys the current buffer and prompts for the name of a buffer to select instead.

---

**Kill Some Buffers (m-X)**

For each existing buffer, tells you something about the status of the buffer and asks whether or not to delete it. If you elect to delete a buffer that has been modified since it was last saved, the command offers to save it first.

---

## **Buffer Commands, cont'd.**

---

### **Kill Or Save Buffers (M-X)**

Puts up a multiple-choice menu listing all existing buffers. You can then choose which buffers to destroy and which to write out to files. This command appears on the editor menu.

---

---

## Appending, Prepending, and Inserting Text

---

### Appending a Region to a Buffer

c-X A

Append To Buffer

Prompts for the name of a buffer and appends the contents of the region onto the end of the specified buffer.

---

### Appending a Region to a File

Append To File (m-X)

Prompts for the name of a file (Append region to end of file:) and appends the contents of the region onto the end of the specified file, writing a new version of that file.

---

### Prepending a Region to a File

Prepend To File (m-X)

Prompts for the name of a file and prepends the contents of the region onto the beginning of the specified file.

---

### Inserting a Buffer Into Another Buffer

Insert Buffer (m-X)

Prompts for the name of a buffer and inserts the entire contents of that buffer into the current buffer at the cursor.

---

### Inserting a File Into a Buffer

Insert File (m-X)

Prompts for the name of a file and inserts the contents of that file into the current buffer at the cursor.

---

---

## Comparing Files and Buffers

---

### Source Compare

#### Source Compare (m-X)

Compares two files or buffers, prompting for type (F or B) and name of each, and displays the results of the comparison in the typeout window. It saves the output in a support buffer named \*Source-Compare-N\*. You can read the comparison while checking the file, for example, by going into two window mode with the comparison in one window and the file in the other.

---

#### Example

This example shows a comparison between the file *new*, as it was read into the buffer, and the buffer *new*, which contains the contents of the file *new* *plus* changes that have been made:

```
Source compare made by ESG on 12/21/83 12:30:40 *-Fundamental*-  
of Buffer new /dass/pubs/pgs VIXEN: with File  
VIXEN: /dass/pubs/pgs/new
```

```
****Buffer new /dass/pubs/pgs VIXEN:, Line #179  
Source Compare Merge compares two files or buffers,  
prompting for type and name, and merges the differences
```

```
****File VIXEN: /dass/pubs/pgs/new, Line #179  
Compares two files or buffers, prompting for type and  
name, and merges the differences
```

```
*****
```

```
Done.
```

---

### Source Compare Merge

#### Source Compare Merge (m-X)

Compares two files or buffers, prompting for type and name, and produces a new version that reconciles the differences between the two. You choose which version (if any) to accept. You can also manually edit one or both versions.

**Comparing Files and Buffers, cont'd.**

At each place where the sources differ, the command prompts you twice. The first time you specify what to do to resolve the difference (prompts: Specify which version to keep:). (For example, you can keep one or the other version, both of them, or neither.) Respond to the prompt using these subcommands:

<i>Option</i>	<i>Action</i>
1	Leaves the first alternative in the text, redisplay the contents, and asks for confirmation of change.
2	Leaves the second alternative in the text, redisplay the contents, and asks for confirmation of change.
*	Leaves both alternatives in the text, redisplay the contents, and asks for confirmation of change.
I	Leaves both alternatives in the text, along with the message lines from the source compare (** MERGE LOSSAGE **), but does not ask for confirmation.
SPACE	Leaves both alternatives in the text, but does not redisplay the contents or ask for confirmation.
!	Disposes of this and all remaining differences the same way, without confirmation. It asks: What to do with remaining differences (1, 2, *, I, or RUBOUT?
c-R	Edits. Press END to return to this question.
RUBOUT	Leaves nothing in the new buffer, does not redisplay the contents or ask for confirmation.

The second time you confirm or reject the change that was made. The screen now shows the change that was made as a result of your choice and prompts: Please confirm the change that has been made: (SPACE, RUBOUT, or c-R). Confirming it keeps that change and moves on to the next difference. Rejecting it returns to the prior appearance so that you can make a different choice:

<i>Option</i>	<i>Action</i>
SPACE	Yes, that's right.
RUBOUT	No, take that back.
c-R	Edit. Press END to return to this question.

When you finish confirming your decisions, Zmacs incorporates all changes into the new version in the specified buffer and the minibuffer displays: Done. Resectionizing the buffer.



---

## Comparing Files and Buffers, *cont'd.*

---

Source Compare Merge also has a mouse interface. You can answer the first question by clicking left on the text you want to keep or on the dividing line between them to keep both. You can answer the second question by clicking left for "yes" (changes confirmed) or middle for "no" (changes rejected).

---

### **Compare/Merge Commands for Definitions**

The following commands operate on definitions by comparing, or comparing and merging, the current version with the newest version, newest version on disk, or installed version.

---

#### *Comparing/Merging Current/Newest Versions*

##### Source Compare Newest Definition (m-x)

Compares the current definition with the newest version in the normal source file for this definition, regardless of patch files.

---

##### Source Compare Merge Newest Definition (m-x)

Compares and merges the current definition with the newest version in the normal source file.

---

#### *Comparing/Merging Current/Saved Versions*

##### Source Compare Saved Definition (m-x)

Compares the current definition with the source for the newest version on disk.

---

##### Source Compare Merge Saved Definition (m-x)

Compares and merges the current definition with the source for the newest version on disk.

---

#### *Comparing/Merging Current/Installed Versions*

##### Source Compare Installed Definition (m-x)

Compares the current definition with the source for the installed version.

---

---

**Comparing Files and Buffers, *cont'd.***

---

**Source Compare Merge Installed Definition (m-X)**

Compares the current definition with the source for the installed version, merging the results.

---

---

## Window Commands

---

### Using Two Windows, Select Bottom

`c-X 2` Two Windows

Shows two windows, selecting the bottom one. It splits the frame into two editor windows, selects the bottom one, and displays the next buffer from the global history in it. With a numeric argument, it displays that same buffer in the second window.

---

### Using Two Windows, Select Top

`c-X 3` View Two Windows

Shows two windows, selecting the top one. It splits the frame into two editor windows, selects the top one, and displays the next buffer from the global history in it. With a numeric argument, it displays that same buffer in the second window.

---

### Two Windows, Specify Other Contents

`c-X 4` Modified Two Windows

Selects a buffer, file, or definition in the other window. `c-X 4` combines the functions of splitting the frame and selecting contents for the second window. It prompts for the type of contents you want for the second window (Select what in other window? (B, F, D, or J), for buffer, file, definition, or jump to register). Then it reads the name of the file, buffer, definition, or register that you want to select for that window.

---

### Two Windows, Region in Top

`c-X 8` Two Windows Showing Region

Makes two windows on the same buffer, with the top one displaying the current region.

---

### Change Window Size

`c-X ^` Grow Window

Changes the size of the current window by some number of lines. With a positive numeric argument, it expands the window; with a negative numeric argument, it shrinks the window.

---

---

**Window Commands, cont'd.**

---

**Choose Other Window**

`c-X 0` Other Window  
Moves the cursor to the other window.

---

**Return to One Window**

`c-X 1` One Window  
Returns the editor frame to displaying only one window. It expands the current window to use the whole frame. With a numeric argument, it expands the other window to use the whole frame.

---

**Scroll Other Window**

`c-M-V` Scroll Other Window  
Scrolls the other window up several lines. By default, it scrolls the same way as `c-V`. With no argument, it scrolls a full screen. With just a minus sign as an argument (`c-M- -V`), it scrolls a full screen backward. A numeric argument tells it how many lines to scroll — a positive number scrolls forward, a negative number scrolls backward.

---

**Split Screen**

Split Screen (`m-X`)  
Pops up a menu that offers to create a new buffer or find a file; makes several windows split among the buffers as specified.

---

---

## File Manipulation Commands

---

### Overview

The commands described in this section are unlike most other Zmacs commands. Their main business is not manipulating buffers and their contents, but rather files out in a file system. First we discuss some commands for dealing with files, then we describe buffer and file attributes, and finally we explain *Dired Mode*, a special Zmacs mode for directory editing.

---

### Listing Files in a Directory

List Files (M-X)

Prompts for the name of a directory and displays the names of all the files in that directory.

The file names are mouse-sensitive. Pointing at a file name and clicking left is just like doing a C-X C-F (Find File) on that file. Clicking right pops up a menu with three items:

- |         |   |
|---------|---|
| Load    | Loads the file into the Lisp world. The file must be either a Lisp source file or a compiled Lisp (" <i>bin</i> " or " <i>qbin</i> ") file. |
| Find    | Reads the file into an editor buffer.   |
| Compare | Compares the file with its most recent version and prints the differences.  |

---

### Displaying the Contents of a Directory

C-X C-D Display Directory

Displays the directory of the file in the current Zmacs buffer. C-X C-D does not ask for a directory but lists files with the same host, device, directory, and name as the file in the current buffer. It lists files with any type and version. With a numeric argument, it prompts for a directory to list and lists that directory.

The heading of the directory listing is mouse-sensitive; clicking left on it selects a Dired buffer containing that directory listing.

C-U C-X C-D does the same thing as List Files, except that it gives more details about each file.

---

---

## File Manipulation Commands, *cont'd.*

---

### Viewing a File

View File is for when you just want to look at a file, not edit it.

View File (m-X)

Prompts for the name of a file and prints out the file contents for viewing only in the typeout window. If there is more than a screenful, it pauses between screenfuls displaying a --MORE-- message at the bottom.

SPACE            Displays the next screenful.

BACKSPACE       Displays the previous screenful.

RUBOUT           Exits.

Anything else exits and is executed as a command.

---

### Viewing the Properties of a File

View File Properties (m-X)

Prompts for the name of a file and displays all the properties of the file that are maintained by the file system on which it resides. These are the properties like creation date and time, author, time of last access, and length. For files on a Lisp Machine file system, it displays user-defined properties as well.

It prompts for a file specification, which it merges with the current default to form the pathname. Wildcards are not accepted; this must correspond to a unique file or directory name.

---

### Hardcopying a File

Hardcopy File (m-X)

Prompts for the name of a file and then prints the specified file on a hardcopy printer.

---

### Renaming a File

Rename File (m-X)

Renames one or more files. It prompts for the name of a file and then asks for a new name for that file. It renames the specified file with that new name.

If the source file specification is wild, the target file specification must also be wild.

---

---

## File Manipulation Commands, *cont'd.*

---

### Copying a File Into Another

#### Copy File (m-X)

Copies any type of file to another specified file.

Prompts from the minibuffer for the names of two files and copies the contents of the first into the second. In file systems supporting multiple versions, this creates a new version of the second file whose contents are identical to those of the first.

Copy File determines whether the source file is a character file or a binary file and copies the file appropriately. Different file systems sometimes use different character sets, and if the file is a character file, character translations have to be done (for example, on some hosts Return characters have to be converted into a carriage return and a line feed).

The numeric argument controls copying of attributes and properties. With no numeric argument, it copies creation date and author and determines the mode (binary or character) of copy by the file being copied. To force mode, or suppress author or creation date copying, supply a numeric argument created by adding the values corresponding to the descriptions below:

- 1 Force copy in 16-bit binary mode.
- 2 Force copy in character (text) mode.
- 4 Suppress copy of author.
- 8 Suppress copy of creation date.

---

### *Examples*

For example, to suppress author and creation date for copying:

c-12 Copy File (m-X)

Use wildcard pathnames to specify groups of files for copying. For example, to copy all files in the subdirectory mine:

```
F:>program>mine>*.*
```

If the source file specification is wild, the target file specification must also be wild.

---

**File Manipulation Commands, cont'd.**

---

```

you type: m-X Copy File
          Zmacs: Copy File from:
you type: scrc:<lmfs>*.*sp;0
          (Copies all the newest .LISP and .LSPs)
          Zmacs: to:
you type: ff:>sys-hold>scrc-sources>old-*.x.*
          Zmacs: SCRC:<LMFS>TEST.LISP.3 is copied into
                  ff:>sys-hold>scrc-sources>old-test.lisp.3

          SCRC:<LMFS>FILES.LISP.147 is copied into
                  ff:>sys-hold>scrc-sources>old-files.lisp.147

```

Note that .LSP gets mapped into .lisp because Copy File uses canonical types when the type of the target pattern is **:wild**. This command can copy file authors and creation dates, when the target operating system supports setting these attributes. This action is not the default.

---

**Creating Links to Files****Create Link (m-X)**

Creates a link to a file. It prompts in the minibuffer for the names of two files as arguments; first the name of the link, then the name of the target pointed to by the link.

---

**Deleting Files****Delete File (m-X)**

Deletes a file. It prompts in the minibuffer for a file name, which can be wild. With a wild name as an argument, deletes multiple files. It lists the files that would be deleted and requires that you confirm the list. It deletes the files, showing any errors that occur but continuing rather than halting. Displays a message in the minibuffer if the specified file does not exist.

---

**Deleting Multiple Versions****Reap File (m-X)**

This command works in file systems supporting multiple versions. It prompts for the name of a file (not including version number) and deletes excess or temporary versions of the specified file, keeping the most recent *n* files. With no numeric argument, the default keeps two versions and deletes any excess. Any numeric argument specifies the number of versions to keep. It prompts for confirmation of files being deleted.

---



---

## File Manipulation Commands, *cont'd.*

---

### Clean Directory (m-X)

Deletes excess versions or temporary file types in the specified directory. The default for excess versions is more than two. It prompts for confirmation of files being deleted. With a numeric argument *n*, it deletes excess versions greater than *n*.

Excess is defined by the value of the Zmacs variable File Versions Kept or by the numeric argument. The temporary file types are defined by the Zmacs variable Temp File Type List. It accepts wildcards in the file name specification. (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193).

---

## Changing the Properties of a File

### Change File Properties (m-X)

Edits the properties of a file. Properties are the qualities of the file that are maintained by the file system on which it resides, such as creation date and time, author, time of last access, and length. For files on a Lisp Machine file system, this means user-defined properties as well. It prompts for the name of a file and pops up a choose-variable-values window, allowing you to alter various properties of the file. The exact properties that can be varied depend on the file system, but they might include:

- Generation (version) retention count
  - Author
  - Creation, modification, and reference dates
  - Protection flags
  - Other file-associated information
- 

## Creating a Directory

### Create Directory (m-X)

Creates a new directory. It prompts for a directory name, using the standard conventions for defaults. For consistency between hierarchical and nonhierarchical file systems, you specify the directory to be created as the directory component of a pathname. That is, you must end the directory name with whatever delimiter or separator is appropriate for the host.

---

---

**File Manipulation Commands, cont'd.**

---

*Example*

<i>Host</i>	<i>Directory string</i>	<i>Result</i>
TOPS-20	<A.B.C>	Creates directory C
Multics	>udd>Sun>Luna>z>	Creates directory z
Lisp Machine	>sun>luna>b>	Creates directory b
UNIX	/usr/jek/new/	Creates directory new

Currently, the file servers for VAX/VMS and TOPS-20 can fail to create directories, due to missing options.

---

---

## Buffer and File Attributes

---

### Attributes

Each buffer and generic pathname has *attributes*, such as Package and Base, which can also be displayed in the text of the buffer or file as an attribute list. An attribute list must be the first nonblank line of a file, and it must set off the listing of attributes on each side with the characters *-\**. If this line appears in a file, the attributes it specifies are bound to the values in the attribute list when you read or load the file.

---

### How They Work

Suppose you want your new program to be part of a package named **graphics** that contains graphics programs. In this case, you want to set the Package attribute to **graphics** in three places: the generic pathname's property list; the buffer data structure; and the buffer text. Here are two ways to make the change:

- If the package already exists in your Lisp environment, use Set Package (*m-x*) to set the package for the buffer. The command asks you whether or not to set the package for the file and attribute list as well. You can use this command to create a new package.
- Use Update Attribute List (*m-x*) to transfer the current buffer attributes to the file and create a text attribute list. Edit the attribute list, changing the package. Use Reparse Attribute List (*m-x*) to transfer the attributes in the attribute list to the file and the buffer data structure. If the package you specify by editing the attribute list does not exist in your Lisp environment, Reparse Attribute List asks you whether or not to create it with default characteristics.

---

### Attribute- Manipulating Commands

#### Update Attribute List (*m-x*)

Updates the attribute list (*-\** line) of the buffer. It creates or updates the attribute list of the file, using the current set of parameters. A new attribute list inherits the Package, Mode, Backspace, and Fonts attributes of the current buffer. It includes the Backspace and Fonts attributes in the line only if they have values other than the defaults. It does not change other attributes in an existing mode line.

---

#### Reparse Attribute List (*m-x*)

Reparses the attribute list (*-\** line) of the buffer. It finds the attribute list for the buffer and processes it to set up the

---

**Buffer and File Attributes, cont'd.**

---

environment that the line specifies. It changes the major mode, package, base, and so on, as necessary. When you edit the attribute list, you should then use this command to make the changes take effect in Zmacs. The changes take effect both for the editor buffer and for the file that the buffer is editing.

*Example*

Suppose the package for the current buffer is **user** and the base is 8. You want to create a package called **graphics** for the buffer and associated file. You also want to set the base to 10. If no attribute list exists, use Update Attribute List (M-X) to create one using the attributes of the current buffer. An attribute list appears as the first line of the buffer:

```
;;; -*- Mode: LISP; Package: USER; Base: 8 -*-
```

Now edit the buffer attribute list to change the package name from USER to GRAPHICS and to change the base from 8 to 10. Use Reparse Attribute List (M-X). The command queries:

```
The file belongs in package GRAPHICS, which does not exist.
Create it with default characteristics,
Try again, or Use another package? (C, T, or U)
```

Answer C to create the new package. The package becomes **graphics** and the base 10 for the buffer and the file.

---

**File Attribute Checking**

Zmacs notes errors in file attribute lists and warns you when it finds an unknown attribute. It goes ahead and ignores the unknown attribute in the list. The purpose of the warning is simply to help you detect misspellings.

---

**Setting the Package**

Set Package (M-X)

Changes the package associated with the buffer. It prompts for a new package, offering to create the package if necessary. Forms that are read from the buffer are read in that package. (The default value for this attribute is **user**.)

You can have any package as the default package by specifying it as the value of the Zmacs variable `Default Package`. (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193). You can set the variable in your `lispm-init.l` file (see "Creating an Init File", page 196) by using the internal form of its name.

## Buffer and File Attributes, *cont'd.*

---

For example, in your init file:

```
(login-forms  
  (setq zwei:*default-package* (pkg-find-package "tv")))
```

If you set the variable to **nil**, it sets the default to the package from the previous buffer.

Information about the package attribute exists in four places. Set Package offers to set the package for the generic pathname attribute list and updates the attribute line in the buffer when you answer Yes to:

Set it for the file and attribute list too?

Your answer affects the various versions of the package attribute as follows:

<i>Location</i>	<i>"Y"</i>	<i>"N"</i>
Generic pathname	changes	same
Buffer property	changes	changes
Buffer text	changes	same
Current package	changes	changes

The system is informed that the file belongs to the specified package. If you are not sure what to answer, say Yes. The global variable **zwei:\*set-attribute-updates-list\*** controls this query. Its default value is **:ask**. Setting the variable to **t** means Yes; **nil** means No.

---

### Other Set commands for File and Buffer Attributes

Each of the file attributes has a Set command associated with it. You have two choices when you want to change an attribute for a file:

- Edit the text of the buffer and then use Reparse Attribute List.
  - Use the relevant Set command and answer Y to its query. The meanings for Y and N are the same as for the Set Package command (except that only the Set Package command affects the current package).
-

---

**Buffer and File Attributes, cont'd.**

---

**Update Attribute  
List Query**

The Set commands use the value of the global variable **zwei:\*set-attribute-updates-list\*** to determine whether to query you about updating the file attribute list. The default value for the variable is **:ask**; set to **nil** to suppress the query.

<i>Value</i>	<i>Meaning</i>
<b>:ask</b>	Always asks whether to update the attribute list.
<b>nil</b>	Never updates the attribute list.
<b>t</b>	Always updates the attribute list.

---

**Set attribute (m-X)**

where *attribute* is one of the following: Backspace, Base, Fonts, Lowercase, Nofill, Package, Patch File, Tab Width, or Vsp. It sets *attribute* for the current buffer. It queries whether or not to set *attribute* for the file and in the text attribute list.

---

**Attribute Descriptions**

The following table describes some of the attributes, their associated Set commands, and the default value for the attribute.

Backspace	The Set Backspace command (default value <b>nil</b> ) controls whether a backspace character in a file displays as the word "back-space" ("overstrike" on an LM-2) with a lozenge around it or performs the backspace. The default is the lozenge form.
Base	The Set Base command (default value 8) specifies the value of <b>ibase</b> that the Lisp reader uses when reading forms from the file. Thus, Base controls the <b>ibase</b> used when you evaluate or compile parts of the buffer, <i>and</i> controls the value of <b>base</b> for printing during evaluating all or part of the buffer. This value does not affect the values of either <b>base</b> or <b>ibase</b> in the Lisp Listener you get by using SUSPEND (BREAK on an LM-2).
Fonts	The Set Fonts command (default value <b>nil</b> ) changes the set of fonts to use. It reads a sequence of fonts names separated by spaces from the minibuffer.

---

## Buffer and File Attributes, *cont'd.*

---

### Lowercase

The Set Lowercase command (default value **nil**) means that the file being edited is intended to contain lowercase code or text. When the Lowercase attribute is **nil** (that is, not present), whatever you want in the way of case handling prevails. People who want automatic uppercase code would use the following in their `lisp-init` file (see also "Creating An Init File", page 196):

```
(login-forms
 (setq zwei:lisp-mode-hook
      'zwei:electric-shift-lock-if-appropriate))
```

When the Lowercase attribute is anything but **nil** (you answer **Y** to its query), the Electric Shift Lock Mode is never turned on automatically.

### Nofill

The Set Nofill command has a default value of **nil**, which means that whatever you want in the way of autofilling behavior prevails. When Nofill is anything else (you answer **Y** to its query), it means that autofilling is not appropriate for people who specify the mode of "autofilling if appropriate".

Use Nofill sparingly. Setting it means that everyone who edits the file has to be satisfied with Auto Fill Mode being off by default. In most cases, it is more reasonable to let an individual user's preferences prevail. It is useful for files that are not plain text, such as mailing lists, where you need to avoid spurious line breaks.

People who want to have autofilling turned on by default should use the following in their `lisp-init` file (see also "Creating An Init File", page 196):

```
(login-forms
 (setq zwei:text-mode-hook
      'zwei:auto-fill-if-appropriate))
```

People who do not want it never get it by default.

---

**Buffer and File Attributes, cont'd.**

---

Patch-File	The Set Patch File command has a default value of <b>nil</b> , which means that the file does not contain patches. When a file is classified as containing patches (you answer <b>Y</b> to its query), <b>fdefine</b> does not warn about functions being redefined during loading. Classifying something as a patch file also affects Edit Definition (which prefers files that are not patches) and <b>defvar</b> (which becomes <b>setq</b> ).
Tab-Width	The Set Tab Width command (default 8 characters) specifies how many spaces the editor uses between "tab stops".
Vsp	The Set Vsp command (default 2 pixels) specifies the vertical spacing (in pixels) between the text lines of an editor window. It specifies the distance between the descenders of one line and the ascenders of the next.

---



## Dired Mode

---

### Overview

There is a special Zmacs mode, called *Dired*, just for doing housekeeping in a directory. In this mode, you see the names of all the files in a directory at once, and can manipulate these files in various ways.

---

### Entering Dired

The following commands specify a directory to manipulate and enter Dired mode.

---

#### Dired (m-X)

Prompts for a wildcard file specification for files contained in the specified directory. The default edits all files in the current directory by specifying wild name, type, and version. You must type the pathname in the form acceptable to your host system.

---

c-X D

Dired

Edits the files in the directory that contains the current file.

With a numeric argument of 1, shows files with the same host, device, directory, and name as the file in the current buffer. It lists files with any type and version.

With a c-U argument, it prompts for a wildcard file specification showing the name of a directory to edit.

---

### The Dired Display

When you go into Dired mode, Zmacs creates a special buffer that contains the names of the files that are under consideration, as well as some auxiliary information pertaining to those files. In a typical Dired buffer, each line describes a single file and lists the following information, from left to right:

- An indicator (D) that shows if the file has been marked for deletion or is already deleted
- The physical volume of the file (on some hosts)
- The name of the file
- The length of the file in blocks (where the length of a block is system-dependent)
- The length of the file in bytes, followed by the byte length in bits, enclosed in parentheses
- ! if the file has not been backed up to tape
- \$ if the file has been marked against reaping
- @ if the file has been marked against deletion
- The file's creation date

---

**Dired Mode**, *cont'd.*

---

- The file's creation time
- The date the file was last referenced, enclosed in parentheses
- The author of the file
- Optionally, the name of the last user to read the file

If there are too many files to be displayed in one screenful, the Zmacs window looks only at one section of the directory at a time (although the buffer does contain the names of all the files).

The files are arranged in alphabetical order by name.

---

**Updating the Display**

Use the Revert Buffer ( $m-x$ ) command (described on page 109) to update a Dired display. After using Dired commands (or native host commands) to perform operations on files in your directory, invoke Revert Buffer, which reexecutes Dired with the default directory name and rereads the updated directory into the buffer.

---

**Dired Commands**

Dired mode has its own command table (comtab) for manipulating the files whose names are displayed. These commands are described in this section. All invocations given in this section are with respect to the Dired comtab and do not apply to regular Zmacs.

You use Dired by moving the cursor around to various lines and then specifying operations to be performed on the file listed on that line (the *current file*, while in Dired Mode).

Most Dired commands schedule some action for the future rather than performing it instantly. For example, when you want to delete a file using Dired, you move the cursor to the line describing that file and type **D**. Rather than deleting the file immediately, Dired *marks the file for deletion*. The deletion actually happens when you leave Dired mode and confirm your request (see "Getting Out of Dired", page 134).

Some of the commands in Dired mode take numeric arguments. You type numeric arguments in exactly the same way as you do in Zmacs proper, except that you do not have to hold the CONTROL key down while typing the argument — just typing the number suffices.

---

## Dired Mode, *cont'd.*

---

### *Command Summary*

The following table summarizes the Dired commands:

<i>Character</i>	<i>Action</i>
RUBOUT	Undeletes file above the cursor.
SPACE	Moves to the next file.
!	Moves to the next file that is not backed up.
\$	Complements the Don't Reap (\$) flag.
.	Describes the attribute list of this file. In text files, this is the <i>-*</i> line of the file. In compiled Lisp files, it includes information about the compilation as well.
.	Changes properties of current file.
@	Complements the Don't Delete (@) flag.
=	Compares this file with the newest version (Source Compare).
A	Queues this file for function application.
C	Copies this file to someplace else.
D	Marks the file for deletion (K, c-D, c-K are synonyms).
E	Edits the file in a buffer, or runs Dired if the line is a subdirectory name.
G	Sets and enforces the generation retention count.
nH	Marks excess versions of the file for deletion (argument means whole directory).
L	Loads the file into Lisp.
nN	Moves to the next file with more than <i>n</i> versions (see the Zmacs variable File Versions Kept). (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193).

**Dired Mode, cont'd.**


---

P	Prints the file on the standard hardcopy device.
Q	Exits. It shows the files marked for deletion and prompts for confirmation. The exit display marks files that have special status, using the following marks: : a link > most recent version \$ file marked for not reaping ! file not backed up
R	Renames this file to something else.
U	Undeletes either the file on the current line or the file on the line above.
V	Views the file without creating a buffer (using View File conventions).
X	Executes an extended command (same as M-X).

---

**Default****Pathnames in Dired**

When the current buffer is a Dired buffer, and you execute an editor command that accepts a file name as an argument, the default file name is the file name that appears on the line of the Dired buffer that point is on.

It makes it easier to do things to the file that you are currently operating on in Dired. For example, you can move point to some line, and then do Compile File (M-X), and it will default to that file name.

**Getting Out of Dired**

Q Dired Exit  
END

Leaves Dired mode. It prints the names of files marked for various actions and gets your final confirmation that these actions are really to be performed.

At this point the available options are:

- Y Delete but do not expunge, also doing any other marked actions.
- N Go back to Dired.
- Q Abort out of Dired (X also works).

## Dired Mode, cont'd.

---

**E** Delete files and expunge directory. This is meaningful for file systems in which there is undeletion, such as TOPS-20, TENEX, and the Lisp Machine file system. This command is useful if you use Dired to free up disk space, since the disk space is not deallocated until the directory is expunged.

Dired Exit performs those actions and returns to the previous buffer.

---

**ABORT**

Dired Abort

Leaves Dired mode at once, without performing any actions on marked files. You can also just switch to another buffer.

---

## Online Documentation for Dired

If you do not have a manual and cannot remember what the commands do, just press **HELP**.

---

**?**

Dired Help

**HELP**

Displays a short table explaining the Dired commands.

---

## Dired Menu

Click right in Dired to display the Dired menu, which offers to perform the following actions on the listing:

- Sort by reference date (up)
- Sort by reference date (down)
- Sort by creation date (up)
- Sort by creation date (down)
- Sort by file name (up)
- Sort by file name (down)
- Sort by file size (up)
- Sort by file size (down)
- Dired Automatic
- Dired Automatic All
- Dired Change File Properties
- Dired Describe Attribute List

Dired Automatic (which includes Dired Automatic All), Dired Change File Properties, and Dired Describe Attribute List are described later in this section.

---



## Dired Mode, *cont'd.*

---

### Viewing and Editing File Contents in Dired

You might want to look at the contents of a file before deciding what to do with it. You might also want to read the file into a buffer and edit it. The following commands provide that capability:

**V** Dired View File

Displays the contents of the current file on the typeout window.

Use this command when you just want to skim the contents of the file, not edit it. You can move forward while viewing with **SPACE** and move backward with **BACKSPACE**.

---

**E** Dired Edit File

Reads the current file into a Zmacs buffer and selects that buffer. You are then back in normal Zmacs and can edit the file normally. When you want to return to Dired mode, just use the **c-m-l** command to reselect the Dired buffer.

---

### Comparing Recent Versions of Files

Often before deciding whether or not to delete a file, you want to find out exactly how extensive the differences are between the file and its most current version. Use the following command:

**=** Dired Srccom

Compares the current file with its most recent version and displays the differences on the typeout window. With an argument of **c-u**, it asks what version to compare it to.

---

### Copying and Renaming Files

**C** Dired Copy File

Copies the current file. It prompts for the new pathname, displaying the default pathname.

**R** Dired Rename File

Renames the current file. It prompts for the new pathname, displaying the default pathname.

---

**Dired Mode, cont'd.**

---

**Marking Files for Deletion**

D Dired Delete  
 K  
 c-D  
 c-K

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

With a numeric argument of  $n$ , it marks the next  $n$  files for deletion.

---

Sometimes you mark a file for deletion by mistake. Here is how you recover from this error:

U Dired Undelete

U takes one of two actions:

1. If the current file is marked for deletion, printing, or a function application (with a D, P, or A), reprieves it.
2. In file systems with soft deletion, U marks a deleted file for undeletion.

In either case, U removes the D, P, or A next to the file. If the current file is not marked with D, P, or A, U reprieves the file on the immediately preceding line, positioning point on that line.

With a numeric argument of  $n$ , it reprieves the files on the next  $n$  lines including the current line.

---

RUBOUT Dired Reverse Undelete

Reprieves the file on the preceding line.

With a numeric argument of  $n$ , it reprieves the files on the previous  $n$  lines including the current line.

---

**Deleting Multiple Versions**

If you are using Dired for housekeeping purposes, the following commands are useful:

N Dired Next Hog

Moves point to the next file with superfluous versions. Superfluous is defined by the value of the Zmacs variable **File Versions Kept** (whose default is 2) or by a numeric argument. (Zmacs variables are described in "How to Specify Zmacs Variable Settings", page 193).

---



## Dired Mode, *cont'd.*

---

H Dired Automatic

This command is also available on the pop-up menu that you get when you click right in Dired. It marks all the superfluous versions of the current file for deletion. With an argument of `c-U`, it marks superfluous versions of all files in the Dired buffer.

---

### Setting Generation Retention Count

G Dired Set Generation Retention Count

Sets and enforces the generation retention count on this group of files, which specifies how many versions to save (that is, deletes multiple versions).

With a numeric argument  $n$ , sets it to  $n$  versions. With no numeric argument, prompts for a number in the minibuffer. An argument of zero means save all versions. *Enforce* means mark for deletion or undeletion.

---

### Protecting Files From Being Reaped

In addition to keeping other users aware of protected files, protection features can also inform the system itself. Some file systems have automatic reaping facilities that go into action when storage becomes scarce. Most such systems have a *don't reap* bit associated with each file; use it to protect only your most vital files.

\$ Dired Complement No Reap Flag

Complements the Don't Reap flag associated with the current file; Dired displays the flag as \$ between the length and date on that line. With a numeric argument of  $n$ , it complements the flag on the next  $n$  files, including the current one.

---

---

**Dired Mode, cont'd.**

---

**Protecting Files  
From Being Deleted**

**@** Dired Complement Dont Delete Flag

Complements the Don't Delete flag associated with the current file; Dired displays the flag as @ between the length and date on that line.

With a numeric argument of  $n$ , it complements the flag on the next  $n$  files, including the current one.

---

**Finding Files That  
Have Not Been  
Backed Up**

Many file systems have tape backup facilities so that files can be copied onto tape against the possibility of a file system disaster. These systems almost always associate a bit with each file that is set when the file is created or modified and cleared when it is backed up to tape.

**!** Dired Next Undumped

Moves point forward to the next file that has not yet been backed up; Dired displays the flag as ! between the length and date on that line.

---

**Marking Files to  
be Hardcopied**

You may want to obtain a hardcopy of a group of related files. Dired allows you to mark files to be hardcopied as well as to be deleted.

**P** Dired Hardcopy File

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

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

---

## Dired Mode, cont'd.

---

### Applying Arbitrary Functions to Files

Very occasionally, you want to perform some operation on selected files in your directory for which there is no Dired command provided. When this occurs, you can write up the operation that you want to perform as a Lisp function, whose single argument is the pathname of the file. The following command is relevant:

A Dired Apply Function

Marks the current file for having an arbitrary function applied to it. Dired puts a A in the first column to show that the file has been so marked. With a numeric argument of  $n$ , it marks the next  $n$  files, including the current one.

---



## Setting the Major Mode

---

## Major Editing Modes

---

### Overview

Whenever you are editing some text, some set of modes is in effect. The buffer is always associated with one major mode that tells the editor what kind of document is being edited. A major mode has the following characteristics:

- It has its own distinct set of key bindings.
- It affects groups of related language-specific items, such as delimiter characters and indentation rules.

The major modes are listed below. You can establish the mode:

- By turning it on using the prefix `m-x` followed by the name of the mode. For example, to invoke Lisp Mode, type: `m-x Lisp Mode`.
- By setting it in the attribute list (see the section "Buffer and File Attributes", page 125)
- By having Zmacs do it for you when you read a file with `c-x c-f`. It recognizes the type component of the pathname of the file (for example, `folon.lisp`) and puts the buffer in the corresponding mode.

---

### Fundamental Mode

Fundamental Mode enters Zwei's fundamental mode (the default mode).

---

### Lisp Mode

Lisp Mode sets things up for editing Lisp code. It puts Indent-For-Lisp on `TAB`.

---

### Text Mode

Sets things up for editing English text. It puts Tab-To-Tab-Stop on `TAB`.

---

### Note

Zmacs supports Fortran Mode as a part of FORTRAN '77, the separately priced software product. For more information, see the *User's Guide to the FORTRAN '77 Tool Kit*.

---

### Macsyma Mode

Macsyma Mode enters a mode for editing Macsyma code. It modifies the delimiter dispatch tables appropriately for Macsyma syntax, makes comment delimiters `/*` and `*/`. It puts Indent-Relative on `TAB`.

---

## Major Editing Modes

---

### Midas Mode

Midas Mode sets things up for editing PDP-10 assembly language code.

---

### Bolio Mode

Bolio Mode sets things up for editing Bolio source files. It is like Text Mode, but also makes `c-m-N`, `c-m-`, and `c-m-*` insert font characters, and makes word-abbrevs for `znll` and `zt`.

---

### Teco Mode

Teco Mode sets things up for editing TECO. It makes comment delimiters be `!*` and `*!`. It puts Indent-Nested on `TAB`, Forward-Teco-Conditional on `m-'`, and Backward-Teco-Conditional on `m-"`.

---

### Pl1 Mode

Pl1 Mode sets things up for editing PL/1 programs. It makes comment delimiters `/*` and `*/`, and puts Indent-For-Pl1 on `TAB`, Roll-Back-Pl1-Indentation on `c-m-H`, and Pl1del on `c-≡`. Underscore is made alphabetic for word commands.

---

### Electric Pl1 Mode

Electric Pl1 Mode sets things up for editing PL/1 programs. It does everything Pl1 Mode does: it makes comment delimiters `/*` and `*/`, puts Indent-for-Pl1 on `TAB`, Roll-Back-Pl1-Indentation on `c-m-H`, and Pl1del on `c-≡`. Underscore is made alphabetic for word commands. In addition, `;` is Pl1-Electric-Semicolon, `:` is Pl1-Electric-Colon, `#` is Rubout, `@` is Clear, `\` is Quoted Insert.

---





## **Changing Case and Indentation**

## Changing Case

---

### Overview

Zmacs offers extended commands that convert the case of the code for words, regions, and buffers.

---

### Changing Case of Words

**m-C** **Uppercase Initial**

Puts next word in lowercase, but capitalizes initial character. With an argument, it capitalizes that many words.

---

**m-L** **Lowercase Word**

Puts next word in lowercase. With an argument, it puts that many words in lowercase.

---

**m-U** **Uppercase Word**

Puts next word in uppercase. With an argument, it puts that many words in uppercase.

---

### Changing Case of Regions

**c-X c-U** **Uppercase Region**

Uppercases the region.

---

**c-X c-L** **Lowercase Region**

Lowercases the region.

---

**Uppercase Code in Region (m-X)**

Converts all code (not comments, strings, or quoted characters) to uppercase. This gives the same effect as retyping that text while in Electric Shift Lock Mode. It operates on the region if there is one, otherwise it operates on the current definition.

---

**Lowercase Code in Region (m-X)**

Converts all code (not comments, strings, or quoted characters) to lowercase. It operates on the region if there is one, otherwise it operates on the current definition.

---

**Changing Case, cont'd.**

---

**Changing Case of Buffers****Uppercase Code in Buffer (M-X)**

Converts all code (not comments, strings, or quoted characters) to uppercase. This gives the same effect as retyping that text while in Electric Shift Lock Mode. It queries for a buffer name (the default is the current buffer) and operates on that buffer.

---

**Lowercase Code in Buffer (M-X)**

Converts all code (not comments, strings, or quoted characters) to lowercase. It queries for a buffer name (the default is the current buffer) and operates on that buffer.

---

## Indentation

---

### Overview

Proper indentation helps make complicated Lisp programs readable. Indentation should reflect the structure of a program. An expression should be indented so that its subforms are easily identifiable, and so that a function can be related to its arguments by eye, without counting parentheses.

The indentation commands work in any Zmacs major mode; the TAB key indents differently depending on the mode. When you give an indent command an argument of  $n$ ,  $n$  equals the number of Space characters in the default font.

---

### Indenting Current Line

TAB

In Lisp mode, the TAB key indents the current line of Lisp code correctly with respect to the line above it. (In most other modes, TAB inserts a Tab character.) Point remains fixed with respect to the code.

With a numeric argument  $n$ , it indents the next  $n$  lines including the current one, and leaves point at the  $n+1$ st line.

---

c-TAB

Indent Differently

Tries to indent this line differently. If called repeatedly, it makes multiple attempts.

---

m-TAB

Insert Tab

Inserts a Tab character, even in Lisp Mode, in the buffer at point.

---

c-n-TAB

Indent For Lisp

Indents this line to make ground (indented) LISP code, even in a mode other than Lisp mode. Numeric argument specifies number of lines to indent.

---

### Centering the Current Line

m-S

Center Line

Centers the text of the current line within the line. With an argument  $n$ , it centers  $n$  lines and moves past them.

---

## Indentation, *cont'd.*

---

### Indenting New Line

The keystroke combination RETURN TAB gets you into the right position to start typing the next line of code. LINE is the abbreviation for that combination.

LINE Indent New Line

If the next two lines are blank, goes to the next line; otherwise, it creates a new blank line following the current one. In any case, it does a TAB on that blank line.

---

### Reindenting Expression

c-m-Q Indent Sexp

Corrects the indentation of the expression following point by adjusting the amount of space before each line in the expression. c-m-Q positions point in front of the incorrectly indented expression. This does not affect the indentation of the current line, but only fixes the indentation of following lines with respect to the current line. Use after modifying an expression.

With a numeric argument of  $n$ , it fixes the indentation of the next  $n$  expressions.

---

### Indenting Region

c-m-\ Indent Region

Indents each line in the region. With no argument, it calls the current Tab command to indent. With an argument of  $n$ , it indents each line  $n$  spaces in the current font.

---

### Going Back to First Indented Character

m-M Back To Indentation  
c-m-M  
m-RETURN  
c-m-RETURN

Positions point before the first nonblank character on the current line.

---

**Indentation, cont'd.**

---

**Indenting Region Uniformly**

c-X TAB

Indent Rigidly

c-X c-I

Shifts text in the region sideways as a unit. All lines in the region have their indentation increased by the numeric argument of the command (the argument can be negative).

---

**Aligning Indentation**

Indent Under (c-m-X)

Indents to align under *string*, which is read from the minibuffer. It searches back, line by line, forward in each line, for a string that matches the one read and that is farther to the right than the cursor already is. It indents to align with the string found, removing any previous indentation first.

---

**Deleting Indentation**

m-^

Delete Indentation

c-m-^

Deletes the newline character and any indentation at the beginning of the current line. It tacks the current line onto the end of the previous line, leaving one space between them when appropriate, for example, at the beginning of a sentence.

With any numeric argument, it moves down a line first, thus killing the end of the current line.

---

**New Line with  
This Indentation**

m-0

This Indentation

Makes a new line after the current one, deducing the new line's indentation from point's position on the current line. If point is to the left of the first nonblank character on the current line, it indents the new line exactly like the current one. But if point is to the right of the first nonblank character, it indents the new line to the current position of point. Regardless, it leaves point at the end of the newly created line.

With a numeric argument, the new line is always indented like the current one, no matter where point is. With an argument of zero, it indents current line to point.

---

**Indentation, cont'd.**

---

**Moving Rest of  
Line Down**

c-m-0

Split Line

Moves rest of current line down one line. It inserts a carriage return and indents new line directly beneath point. With a numeric argument *n*, it moves down *n* lines.

---

**Inserting Blank Line**

c-0

Make Room

Inserts a blank line after point. With a numeric argument *n*, it inserts *n* blank lines.

---

**Deleting Blank Line**

c-X c-0

Delete Blank Lines

Deletes any blank lines around the end of the current line.

---





## **Editing Lisp Programs**

## Introduction

---

Lisp Machine programmers develop programs in repeated cycles, each a sequence of editing, compiling, testing, and debugging. These cycles are often nested. Zmacs allows you to edit and test large programs dynamically, without frequent file system operations. This manual does not describe any style of interacting with the environment in developing Lisp programs. However, the *Programming Development Tools and Techniques* manual shows just that: It focuses on the interaction between programmers and the Lisp Machine, presenting ways of using helpful Lisp Machine features and tools during each stage of program development.

As a programmer on a Lisp Machine you typically read a file containing Lisp code into an editor buffer, make modifications, test the results, make more changes, and so on, until satisfied with the behavior of the program. Only then do you need to write the buffer back out to the file system. The debugging loop is much tighter and more responsive than in traditional programming environments. You can even evaluate Lisp forms directly from inside the editor, without returning to a Lisp Listener. Alternatively, you can divide the screen into a Lisp Listener window and a Zmacs window, so that you can direct your attention to either without changing the display.

Zmacs provides extensive features for locating source code of specified functions. If an error occurs, the Debugger can cause Zmacs to read in the source of the function that got the error. You can then debug and recompile the function. Similar features complement the message-passing capabilities of the Zetalisp language.

When you edit a file whose file type is "lisp", Zmacs puts that buffer into Lisp mode. A command exists for explicitly placing a buffer in Lisp mode:

Lisp Mode (M-X)

Lisp Mode

Places the current buffer into Lisp mode.

---



**Commenting Code, cont'd.**

---

**Moving Down to  
Comment on Next Line****m-N** **Down Comment Line**

Moves point to the beginning of the comment on the next line. If there is no comment on the next line, it creates one. If the comment on the current line is empty, it deletes it before going to the next line.

With a numeric argument  $n$ , it moves point to the beginning of the comment on the  $n$ th line after the current one.

---

**Moving Up to  
Comment on  
Previous Line****m-P** **Up Comment Line**

Moves point to the beginning of the comment on the previous line. If there is no comment on the previous line, it creates one. If the comment on the current line is empty, deletes it before going on to the previous line.

With a numeric argument  $n$ , it moves point to the beginning of the comment on the  $n$ th line before the current one.

---

**Setting the  
Comment Column****c-X ;** **Set Comment Column**

Sets the comment column to be the current horizontal position of the cursor.

With a numeric argument, it finds the nearest comment above the current line, sets the comment column to line up with that comment, and actually puts a comment on the current line at that column.

---

**Creating a New  
Indented  
Comment Line****m-LINE** **Indent New Comment Line**

Makes a new blank line after the current line and starts a new comment there, indented properly. If there was already a comment on the current line, the comment on the new line is of the same kind. That is, it has the same number of semicolons and is

## Commenting Code, *cont'd.*

---

indented the same. If there was no comment on the starting line, `m-LINE` starts a new line, indenting the new line as appropriate for the major mode.

---

## Commenting Regions

`c-x c-;`

Comment Out Region

Comments out each of the lines in the region. When the region ends at the beginning of a line, it does not comment out that line. If any part of the line is part of the region, then it does comment out that line.

A numeric argument activates lines in the region that have been commented out. When any part of the line is part of the region, it removes commenting from around that line. This assumes that any comment starting in column 1 is fair game. It stops when it encounters a line that does not begin the way a comment would, even if more lines that have been commented out remain in the region. It does keep the remainder of the region in this case, so that you can resume.

---

Uncomment Region (`m-x`)

Removes all comments from lines whose beginnings are contained in the region.

---

---

## Evaluation and Compilation

---

### Overview

The commands in this section form a link between the Zmacs editor and the Lisp language. They allow the evaluation and compilation of code from Zmacs buffers. These commands are an important part of the debugging loop.

When a Lisp form is being compiled or evaluated, the editor displays a message that classifies what is being compiled.

It classifies macros as functions (because these go in the function cell of a symbol). For example:

```
Compiling Function SUN
Evaluating Variable MARS
Compiling Flavor STAR
```

---

### Evaluation

**m-ESCAPE**

Evaluate Minibuffer

Evaluates expressions from the minibuffer. You enter Lisp expressions in the minibuffer, which are evaluated when you press **END**. The value of the expression itself appears in the echo area. If the expression displays any output, that appears as a typeout window.

---

**Evaluate Into Buffer (m-X)**

Evaluates an expression read from the minibuffer and inserts the result into the buffer. You enter a Lisp expression in the minibuffer, which is evaluated when you press **END**. The result of evaluating the expression appears in the buffer before point. With a numeric argument, it also inserts any typeout that occurs during the evaluation into the buffer.

---

**Evaluate Buffer (m-X)**

Evaluates the entire buffer. The result of evaluating the buffer appears in the minibuffer. With a numeric argument, it evaluates from point to the end of the buffer.

---

**Evaluate Region (m-X)**

**c-sh-E**

Evaluates the region. When no region has been defined, it evaluates the current definition. It shows the results in the echo area.

---

---

## Evaluation and Compilation, *cont'd.*

---

**c-m-sh-E**

**Evaluate Region Verbose**

Evaluates the region. When no region has been defined, it evaluates the current definition. It shows the results in a typeout window.

---

**Evaluate Region Hack (m-X)**

Evaluates the region, ensuring that any Lisp 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 Changed Definitions (m-X)**

Evaluates any definitions that have changed in any of the current buffers. With a numeric argument, it prompts individually about whether to evaluate particular changed definitions (the default evaluates all changed definitions).

---

**Evaluate Changed Definitions of Buffer (m-X)**

**m-sh-E**

Evaluates any definitions that have changed in the current buffer. With a numeric argument, it prompts individually about whether to evaluate particular changed definitions (the default evaluates all changed definitions).

---

**Evaluate And Replace Into Buffer (m-X)**

Evaluates the Lisp object following point in the buffer and replaces it with its result.

---

**c-m-Z**

**Evaluate And Exit**

Evaluates the buffer and exits Zmacs. It selects the window from which the last **ed** function or the last debugger **c-E** command was executed.

---

---

**Evaluation and Compilation, cont'd.**

---

**Compilation****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 of the current buffers. With a numeric argument, it prompts individually about whether to compile particular changed definitions (the default compiles all changed definitions).

---

**Compile Changed Definitions of Buffer (m-X)****m-sh-C**

Compiles any definitions that have changed in the current buffer. With a numeric argument, it prompts individually about whether to compile particular changed definitions (the default compiles all changed definitions).

---

**Compile File (m-X)**

Compiles a file, offering to save it first (if it has an associated buffer that has been modified). It prompts for a file name in the minibuffer, using the file associated with the current buffer as the default.

---

**Load File (m-X)**

Loads a file, possibly saving and compiling it first. It prompts for a file name, taking the default from the current buffer.

---

**m-Z****Compile And Exit**

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

---



---

## Evaluation and Compilation, *cont'd.*

---

### Compiler Warnings

Compiler warnings are kept in an internal database that you can inspect and manipulate in various ways with several editor commands.

---

#### Compiler Warnings (m-X)

Creates the compiler warnings buffer (called \*Compiler-Warnings-1\*) if it doesn't exist, puts all outstanding compiler warnings in that buffer, and switches to that buffer. You can peruse the compiler warnings by scrolling around and doing text searches through them (see Edit Compiler Warnings).

---

#### Edit Compiler Warnings (m-X)

Prompts you with the name of each file mentioned in the database, allowing you to edit the warnings for that file. It then splits the Zmacs frame into two windows: the upper window displays a warning message and the lower one displays the source code whose compilation caused the warning. After you have finished editing each function, c-. gets you to the next warning: the top window scrolls to show the next warning and the bottom window displays the function associated with this warning. Successive c-.s take you through all of the warning messages for all of the files you specified. When you are done, the last c-. puts the frame back into its previous configuration.

---

#### Edit File Warnings (m-X)

Asks you for the name of the file whose warnings you want to edit. You can give either the source file or the compiled file. Only warnings for this file are edited. If the database does not have any entries for the file you specify, the command prompts you for the name of a file that contains the warnings, in case you know that the warnings are stored in another file.

---

#### Load Compiler Warnings (m-X)

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.

---

## Parenthesizing Expressions

---

m-(

Make ()

Inserts matching parentheses, leaving point between them. With a numeric argument  $n$ , it encloses the next  $n$  Lisp expressions in parentheses. When the number of expressions requested cannot be satisfied, it beeps and does nothing. With point on the open parenthesis of a defun, an argument of 1 encloses the whole defun within a new set of parentheses. Any argument larger than 1 would have no effect. In text mode, a word or a phrase within parentheses is treated as a Lisp form.

See also the description of the command `m-)`, in "Motion Among Top-Level Expressions", page 55.

---

## Expanding Expressions

---

The following editor commands allow you to expand macros.

---

c-sh-M

Macro Expand Expression

Reads the Lisp expression following point and expands the form itself but not any of the subforms within it. It displays the result in the typeout window. With a numeric argument, it pretty-prints the result back into the buffer immediately after the expression.

---

Macro Expand Expression All (M-X)

Reads the Lisp expression following point, and expands all macros within it at all levels. It displays the result in the typeout window. With a numeric argument, it pretty-prints the result back into the buffer immediately after the expression. It assumes that every list in the expression is a form; so if car of a list is a symbol with a macro definition, the purported macro invocation is expanded. (Some sublists are not intended to be Lisp forms, and attempts to macro-expand them are incorrect and may not work.)

---

---

## Locating Source Code to Edit

---

### Introduction

The functions that make up a program or system can depend on each other in complicated ways. When you are editing one function, you sometimes have to go off and look at another function, and possibly modify that one too.

This section describes the Edit Definition command and other commands that list and/or edit various sets of definitions. In addition, two pairs of List and Edit commands help identify changed code by finding or editing *changed* definitions in buffers. By default, the *changed* commands find changes made since the file was read; use numeric arguments to find definitions that have changed since they were last compiled or saved.

---

### The Edit Definition Commands

Edit Definition (*m-*) is a powerful command to find and edit function definitions, macro definitions, global variable definitions, and flavor definitions. In general, Zmacs treats as a definition any top-level expression having in functional position a symbol whose name begins **def**.

It is particularly valuable for finding source code, including system code, that is stored in a file other than that associated with the current buffer. It finds multiple definitions when, for example, a symbol is defined as a function, a variable, and another type of object. It maintains a list of these definitions in a support buffer.

---

*m-*

Edit Definition

Prompts for the name of any named Lisp object. Selects a buffer containing that definition, reading in the source file if necessary, and positions the cursor in front of the definition. With a numeric argument, it edits another definition of the object now being edited. This can be repeated until there are no more definitions of that object.

The prompt has three helpful features: selection by mouse, context default, and completion (for definitions already in the buffer). You can specify a definition by typing the name into the minibuffer or clicking left on a name already in the buffer. If you just press END instead of typing a function name, Zmacs assumes that the function you want is the one at the front of the innermost expression containing point. This default is displayed with the prompt.

## Locating Source Code to Edit, *cont'd.*

---

Zmacs finds definitions this way:

- If the definition is in the current buffer, it moves point there.
- If the definition is in a different buffer, it changes buffers to get to the definition and moves point there.
- If the definition is in a file that has not been read into a Zmacs buffer, Zmacs goes out to the file system to get it, creating a new buffer and reading in the file, and then moves point to the definition.

When a symbol has more than one definition (for example, **list** might be defined both as a function and as a global variable), Zmacs finds all the definitions, but only presents the first one for editing. Zmacs remembers the other definitions, and tells you about them with a message in the echo area. When you have finished with the first definition, you can look at the next by invoking `m-` with a numeric argument. Each time you do this, you bring up a new definition to be edited, until you run out of definitions. `m-` displays **No more definitions** if you try to continue.

---

### *Example*

Suppose you are modifying a function called **sun**, which was written originally by someone else. **sun** calls the unfamiliar **luna**, and you need to find out what **luna** does before proceeding. Use `m-` to peek at the definition of **luna**.

When you type `m-`, Zmacs prompts you for the name of a definition. If point is right in the expression where **luna** is called, the default name is **luna**, and you only need to press **END**. If, on the other hand, point is somewhere else and the default is wrong, you can point at the word **luna** with the mouse or you can type it in. To let you know that you can define a name with the mouse, the mouse cursor changes to an arrow pointing straight up. All the symbols that are names of definitions you could specify become mouse-sensitive.

---

### Edit Changed Definitions (`m-X`)

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 since the buffer was read in and selects the first one on the list. Use `c-` (**Next Possibility**) to move to subsequent definitions.

---

**Locating Source Code to Edit**, *cont'd.*

---

Edit Changed Definitions accepts a numeric argument to control the time point for determining what has changed:

*Value Meaning*

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

**Edit Changed Definitions of Buffer** (*m-X*)

Determines which definitions in the current 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.

Edit Changed Definitions of Buffer accepts a numeric argument to control the time point for determining what has changed:

*Value Meaning*

- 1 Since the file was last read (the default).
  - 2 Since the buffer was last saved.
  - 3 Since the definition was last compiled.
- 

**The List  
Definition Commands****List Definitions** (*m-X*)

Displays the definitions in 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 Changed Definitions** (*m-X*)

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.

---

## Locating Source Code to Edit, *cont'd.*

---

List Changed Definitions accepts a numeric argument to control the time point for determining what has changed:

*Value Meaning*

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

List Changed Definitions of Buffer (M-X)

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.

List Changed Definitions of Buffer accepts a numeric argument to control the time point for determining what has changed:

*Value Meaning*

- 1 Since the file was last read (the default).
  - 2 Since the buffer was last saved.
  - 3 Since the definition was last compiled.
- 

## The Edit Callers Commands

When you are modifying a large system, you often have to make sure that changing a function does not render unusable other functions that call the modified one. Zmacs provides facilities for editing the sources of all the functions defined in the current world that call a given one. This removes some of the unpleasantness of making incompatible changes to large programs and is a good example of how Zmacs interacts with the Lisp environment to make programming easier.

---

Edit Callers (M-X)

Prepares for editing all functions that call the specified one. The prompt is the same kind that Edit Definition gives you. 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

---

**Locating Source Code to Edit** *cont'd.*

---

argument. With an argument of `c-U`, it searches all packages; with `c-U c-U`, it prompts for the name of a package to search. It selects the first caller; use `c-`. (Next Possibility) to move to a subsequent definition.

---

**Multiple Edit Callers** (`m-X`)

Prompts for the names of a group of functions and edits those functions in the current package that call *any* of the specified ones. It reads a function name from the minibuffer, with completion, initially offering a default function name. It continues prompting for more function names until you end the list with RETURN.

By default, it searches the current package. You can control the package being searched by giving the function an argument. With an argument of `c-U`, it searches all packages. With two `c-Us`, it prompts for the name of a package.

---

**List Callers** (`m-X`)

Prompts for the name of a function exactly the way Edit Callers does, but instead of editing the callers in the current package of the specified function, it simply displays their names. The names are mouse-sensitive. If you point at one and click left, you can edit the source of that caller. If you click right, a menu pops up that offers to give the argument list of the selected caller, to disassemble it, to edit it, or to see its documentation string. In addition, `c-`. (Next Possibility) works in this context, offering the first caller to be edited, and queuing up the other callers to be edited in sequence.

With an argument of `c-U`, it lists all the callers in every package. With two `c-Us`, it prompts for the name of a package to search.

---



**Locating Source Code to Edit**, *cont'd.*

---

**Multiple List Callers (M-X)**

Lists all the functions that call the specified functions. It reads a function name from the minibuffer, with completion. It continues prompting for more function names until you end the list with RETURN.

The list of function names is mouse-sensitive: see List Callers (M-X). c-. (Next Possibility) edits the callers.

By default, it searches the current package. You can control the package being searched by giving the function an argument. With an argument of c-U, it searches all packages. With two c-Us, it prompts for the name of a package.

---

## Patching

---

### Introduction

During a typical maintenance session you might make several edits to a system's source files. The patch facility allows you to copy these edits into a patch file so that they can be automatically incorporated into the system to create a new minor version. Edits in a patch file can be of varying levels of complexity — modified function definitions, new functions, modified **defvars** and **defconsts**, or arbitrary forms to be evaluated, even including **loads** of new files. (For complete information about patching, see the section "Patching" in the document *Maintaining Large Systems*.)

---

### The Patch Commands

Start Patch (m-X) and Start Private Patch (m-X) are two commands for initiating a patch.

#### Start Patch (m-X)

Starts a new patch but does not move any Lisp forms into the patch file. Prompts you for the system you want to patch; it must be a system currently loaded. It allocates a new minor version number for that particular system and starts constructing the patch file in an editor buffer.

While you are making your patch file, the minor version number that has been allocated for you is reserved so that nobody else can use it. Thus, if two people are patching a system at the same time, they cannot both get the same minor version number. Also note that you can put together patches for only one system at a time.

If you do a subsequent patch after finishing the current patch (see Finish Patch (m-X)), Start Patch (m-X) asks you which system you wish to patch and starts a new minor version.

---

#### Start Private Patch (m-X)

Similar to Start Patch (m-X), but it does not have any relationship to systems, major and minor version numbers, and official patch directories. Instead of prompting for a system, it prompts for a file name. You can use other patching commands, like Add Patch (m-X), Finish Patch (m-X), and Abort Patch (m-X). When you finish the patch it is written out to the specified file.

This command allows you to make a private patch file that you can load, test, and share with other users before you install it as a numbered patch that all users automatically get.

---

## Patching, *cont'd.*

---

If you change a function, you should recompile it and test it; then, once it works, use Add Patch (m-X), Add Patch Changed Definitions (m-X), or Add Patch Changed Definitions of Buffer (m-X) to put the code in the patch file.

### Add Patch (m-X)

Adds the region (if there is one) or else the current definition to the patch file currently being constructed. If you mistakenly use the command on code that does not work, select the buffer containing the patch file and delete it. Then later you can use Add Patch (m-X) on the corrected version.

---

### Add Patch Changed Definitions of Buffer (m-X)

Selects each definition that was changed in the buffer and asks you whether or not you want the definition patched.

For each definition, you can respond as follows:

<i>Response</i>	<i>Action</i>
-----------------	---------------

Y	Patches the definition.
N	Skips the definition.
P	Patches the definition and any additional definitions in the same buffer without asking any more questions.

A definition needs to be patched if it has been changed since it was last patched or if it has not been patched since the file was read into the buffer.

Note that patching any region of text lying entirely within a definition (with Add Patch (m-X)) counts as patching that definition.

---

### Add Patch Changed Definitions (m-X)

Selects a buffer in which definitions were changed and asks whether or not you want to patch the changed definitions. Answering N skips the buffer and proceeds to the next buffer, if any. Answering Y selects each definition that has changed in that buffer and asks you whether or not you want the definition patched.

---

**Patching, cont'd.**

---

For each definition, you can respond as follows:

*Response Action*

- Y Patches the definition.
- N Skips the definition.
- P Patches the definition and any additional definitions in the same buffer without asking any more questions; when done, it proceeds to the next buffer.

If there are more buffers containing definitions to be patched, it asks questions again when it gets to the next buffer.

A definition needs to be patched if it has been changed since it was last patched or if it has not been patched since the file was read into the buffer.

Note that patching any region of text lying entirely within a definition (with Add Patch (m-X)) counts as patching that definition.

---

After making and testing all of your patches, use Finish Patch (m-X).

**Finish Patch (m-X)**

Installs the patch file so that other users can load it. This compiles the patch file if you have not done so yourself (patches are always compiled). It prompts you for a comment describing the reason for the patch; **load-patches** and **print-system-modifications** print these comments.

Sometimes you start making a patch file and for a variety of reasons do not install it — for example, you decide to abort the patch, you need to end your work session at this machine, or your machine crashes.

---

**Abort Patch (m-X)**

Deallocates the minor version number that was assigned by Start Patch (m-X). It tells Zmacs that you are no longer currently making a patch; the next time you do Start Patch (m-X), Zmacs starts a new patch instead of appending to the one in progress.

---

**Patching**, *cont'd.*

---

**Resume Patch (m-x)**

Allows you to go back to a patch that you were not able to finish in the same session in which you started it. This command works only if you have previously saved all modified buffers.

If the system crashes, use Resume Patch (m-x) and then Abort Patch (m-x). Begin the patch again.

---



## **Customizing the Zmacs Environment**

## Overview

---

### Introduction

Now that you are familiar with the basic Zmacs concepts and techniques, you can set up a large set of minor modes, Zmacs and Lisp variables, and parameters to change the way the editor works. Zmacs's flexibility allows you to change which keys are connected to which commands, write your own commands, and install them in lieu of the standard system commands. A few users make extremely radical changes to the point where almost every key has a new meaning.

This section describes:

- Zmacs minor and major modes, and how they provide a degree of customization
  - Creating new commands with keyboard macros
  - Setting key bindings
  - Specifying Zmacs variable settings
  - Sample init file forms for automatically reloading your customized environment
-



---

## Built-in Customization — Zmacs Minor Modes

---

### Definition — Minor Modes

A *minor mode*:

- Is an option.
  - Is independent of other minor modes and of the selected major mode.
- 

### How It Works

Zmacs has an extended command for each minor mode (*m-X*) that turns the mode on or off. With no argument, the command turns the mode on if it was off and off if it was on. This is known as *toggling*. A positive argument always turns the mode on, and a zero argument or a negative argument always turns it off. All the minor mode commands are suitable for connecting to single- or double-character commands if you want to enter and exit a minor mode frequently. (See the section "Key Bindings", page 191.)

---

### Example — Filling Text

#### Auto Fill Mode (*m-X*)

Turns on *Auto Fill Mode*, a minor mode that inserts Return characters automatically to break lines as you type. You can turn Auto Fill Mode on regardless of your major mode. If the mode line displays *Fill*, Auto Fill Mode is on. If Auto Fill Mode is already turned on, this command turns it off.

This mode is useful when you are typing large amounts of text. It makes it unnecessary to look at the screen or to worry about line length: you just type in the text without newlines and Zmacs inserts them whenever they are needed.

Auto Fill Mode works by establishing a hook that runs after you press one of the activation characters (SPACE, RETURN, ., ?, !, or ") that activate filling in this mode. When you press one of these characters in Auto Fill Mode, Zmacs does more than simply insert it. First it checks to see whether the line exceeds the maximum allowable line length or *fill column* (see Set Fill Column below). If the line is too long, Zmacs finds the last word on the current line that fits inside the fill column. Zmacs then inserts a newline right after that word. Extra spaces (if any) are deleted from the beginning of the newly formed line.

Because of the way Auto Fill Mode works, you will often find yourself typing a word out beyond the fill column. The word will be moved to the next line as soon as you press one of the activation characters.

---

**Built-in Customization — Zmacs Minor Modes, cont'd.**

---

The fill column is used by Auto Fill Mode (and by the paragraph adjusting commands) to decide where to break lines. It is measured in pixels, not in characters, so that Auto Fill Mode works even if characters of different widths appear in a buffer. (A *pixel* is a tiny rectangular area on the screen that is either all white or all black. Pixels are the smallest addressable region of the display. If you look closely, you can see the separate rectangular pixels that make up everything on the display.)

---

You can change the fill column with the following command:

`c-X F` Set Fill Column

Changes the fill column to match up with the current position of the cursor. That means that if point is at the end of a line, filled lines will not be longer than the current one from now on.

With a positive numeric argument  $n$  less than 200, the fill column is set to be  $n$  character-widths, and if  $n$  is 200 or greater, the fill column is set to be  $n$  pixels.

---

### Summary of Minor Modes

---

#### Atom Word Mode (`m-X`)

Makes word-moving commands, in Lisp mode, move over Lisp atoms instead of words. This command does not display anything in the mode line.

---

#### Auto Fill Lisp Comments Mode (`c-m-X`)

Turns on auto filling of comments, but not code. This command displays `Fill-Comments` in the mode line.

---

#### Auto Fill Mode (`m-X`)

Turns on auto filling. Auto Fill mode allows you to type text endlessly without worrying about the width of your screen. Return characters are inserted where needed to prevent lines from becoming too long. This command displays `Fill` in the mode line.

---

#### Electric Font Lock Mode (`m-X`)

Puts comments in font B. This command displays `Electric Font-lock` in the mode line.

---

---

## Built-in Customization — Zmacs Minor Modes, *cont'd.*

---

### Electric Shift Lock Mode (M-X)

Facilitates typing in programs that are in uppercase. Whenever you type a character that is part of a Lisp symbol, such as the name of a function, variable, or special form, Zmacs inserts it in uppercase, but when you type a character that is part of a character string or a comment or after a slash, Zmacs inserts it normally. This command displays Electric Shift-lock in the mode line.

---

### EMACS Mode (M-X)

Provides commands for EMACS users. It puts bit-prefix commands on ESCAPE (ALTMODE on an LM-2), C-^, and C-C, and Universal argument on C-U. It also makes C-I a synonym for TAB, C-H a synonym for BACKSPACE, and C-] a synonym for ABORT. This command displays EMACS in the mode line.

---

### Overwrite Mode (M-X)

Turns on overwrite mode. In overwrite mode, ordinary printing characters replace existing text, instead of inserting themselves next to it. It is good for editing pictures. This command displays Overwrite in the mode line.

---

### Word Abbrev Mode (M-X)

Allows you to define word abbreviations that expand as you type them. This command displays Abbrev in the mode line.

---

*See...*

See "Setting Mode Hooks", page 197, for information about setting minor modes permanently.

---

## Major Modes

---

### User-Defined Major Modes

In Zmacs, you can define your own major modes (see **zwei:defmajor** in the code).

---

### File Types and Major Modes

You can control the default major mode associated with a particular file type. For example, Zmacs sets the major mode to Lisp for files with type `lisp`. The repository for this information is a list called **fs:\*file-type-mode-alist\***. For example, suppose you wanted to associate the file type `tex` with `text` mode:

```
(push '("tex" . :text) fs:*file-type-mode-alist*)
```

The **car** of an element should be either a canonical type symbol or a string when the type is not one of the known canonical types.

In addition, suppose you have files that would require Scribe mode, if Zmacs had such a thing. You can define a correspondence between two major modes, using a global variable called **zwei:\*major-mode-translations\***. It is an alist of major mode names, expressed as keyword symbols. Example:

```
(push '(:scribe . :text) zwei:*major-mode-translations*)
```

---

---

## Creating New Commands: Keyboard Macros

---

### Definition

A *keyboard macro* is a command that you define to abbreviate a sequence of other commands. If you discover that you are about to type `c-N c-D` 40 times, you can define a keyboard macro to do `c-N c-D` and call it with a repeat count of 40.

---

### How It Works

You define a keyboard macro by telling Zmacs that you are about to write a macro and then typing the commands that are the definition. That is, as you are defining a keyboard macro, the definition is being executed for the first time. When you are finished, the keyboard macro is defined and also has been, in effect, executed once. You can then do the whole thing over again by invoking the macro.

---

### Procedure

1. To start defining a keyboard macro, type `c-X (` (Start Kbd Macro). From then on, your commands continue to be executed, but also become part of the definition of the macro. Macro-level: 1 appears in the mode line.
2. If you want to perform an operation on each line, do one of the following:
  - Start by positioning point on the line above the first one to be processed and then begin the macro definition with a `c-N`
  - Start on the proper line and end with a `c-N`.Either way, repeating the macro operates on successive lines.
3. After defining the body of the macro, you can terminate it in several ways.
  - `c-X )` (End Kbd Macro) terminates the definition.
  - An argument of zero to `c-X )` automatically repeats the macro (upon termination of the definition) until it gets an error or reaches the end of the buffer.
  - `c-X )` can be given a repeat count as a numeric argument, in which case it repeats the macro that many times right after defining it, but defining the macro counts as the first repetition (since it is executed as you define it). (Subsequent invocations ignore the numeric argument contained in the macro.)

---

### Example

Inserting an argument of 5 before ending the macro (`...c-5 c-X )`) executes the macro immediately four additional times.

---

**Creating New Commands: Keyboard Macros, cont'd.**

---

**Start Keyboard Macro**

c-X ( Start Kbd Macro  
 Begins defining a keyboard macro. A numeric argument means  
 append to the previous keyboard macro.

---

**End Keyboard Macro**

c-X ) End Kbd Macro  
 Terminates the definition of a keyboard macro.

---

**View Keyboard Macro**

To see the keyboard macro, use View Kbd Macro (m-X), which  
 prints the macro at the top of your screen.

View Kbd Macro (m-X)

Displays the specified keyboard macro. The name of the macro is  
 read from the minibuffer; just RETURN means the last one defined,  
 which can also be temporary.

---

**Call Last  
Keyboard Macro**

The macro thus defined can be invoked again with c-X E (Call Last  
 Kbd Macro), which can be given a repeat count as a numeric  
 argument to execute the macro many times.

c-X E Call Last Kbd Macro  
 Repeats the last keyboard macro.

---

*Example*

The example below defines a keyboard macro that goes to the  
 beginning of a line, inserts a semicolon, and goes to the next line.  
 It also executes the macro four times, including once as it is being  
 defined.

```
c-X (
c-A
;
c-N
c-4 c-X )
```

---

---

## Creating New Commands: Keyboard Macros, *cont'd.*

---

### Defining an Interactive Keyboard Macro

Within the keyboard macro definition, you can specify steps at which you want the macro to query. To define an interactive keyboard macro, use the Kbd Macro Query command after beginning the macro definition (with Start Kbd Macro). Invoke Kbd Macro Query at each spot in the macro where you want the macro to query. Then close the definition with End Kbd Macro.

`c-X Q` Kbd Macro Query

Allows user interaction on each iteration of macro, similar to Query Replace (`m-X`). While defining a keyboard macro, press `c-X Q` at each step where you want a pause to occur. Upon execution of the macro, it stops and waits at each of those steps for one of the following characters:

SPACE	Continues execution of the macro.
RUBOUT	Skips rest of keyboard macro (use nested <code>c-X (</code> and <code>c-X )</code> for grouping to control range of skip).
? or HELP	Displays HELP information.
.	Continues but does not iterate anymore.
!	Continues, iterates, but does not ask anymore.
<code>c-R</code>	Enters editing mode; <code>c-m-FUNCTION R (MACRO R</code> on an LM-2) resumes the keyboard macro.

---

### Naming a Keyboard Macro

Having defined a keyboard macro, you can name it with Name Last Kbd Macro (`m-X`). A prompt (Name for macro:) appears in the minibuffer.

Name Last Kbd Macro (`m-X`)

Assigns a name to the most recent temporary keyboard macro, making it permanent. The new name for the macro is read from the minibuffer.

---

---

## Creating New Commands: Keyboard Macros, *cont'd.*

---

### Using Keyboard Macros to Sort

You can use a keyboard macro to set up a sorting mechanism and run it on any region of text. See the description of Sort Via Keyboard Macros, page 100.

---

### Installing a Macro on a Key

To bind the macro to the key of your choice, use Install Macro (m-X). You are asked to identify the macro and specify the key(s) to which you want it bound.

#### Install Macro (m-X)

Installs a specified user macro on a specified key. The name of the macro is read from the minibuffer, and the keystroke on which to install it is read in the echo area. If the key is currently holding a command prefix (like c-X), it asks you for another character, so that you can redefine c-X commands. However, with a numeric argument, it assumes you want to redefine c-X itself, and does not ask for another character.

---

### Installing a Mouse Macro

You can bind the macro to a mouse click instead of a key using Install Mouse Macro (m-X). This command works similarly to Install Macro.

#### Install Mouse Macro (m-X)

Installs a specified user macro on a specified mouse click. The name of the macro is read from the minibuffer, and the mouse click on which to install it is read in the echo area. When the mouse is clicked to invoke this macro, the macro is invoked from the current location of the mouse cursor.

---

### Deinstalling a Macro

To remove the macro from that key, use Deinstall Macro (m-X). The key is rebound to the standard system usage, if any.

#### Deinstall Macro (m-X)

Deinstalls a keyboard macro.

---



---

## Creating New Commands: Keyboard Macros, *cont'd.*

---

### *Example*

This example shows how to install a macro and deinstall the same macro:

```
you type:      m-X Install Macro
minibuffer:    Name of macro to install (CR for last macro defined):
you type:      macro-name or CR
minibuffer:    Key to get it:
you type:      h-T
```

A menu appears and asks you in which comtab to install the macro:

- Just this editor
- Zmacs
- Zwei

Click on your choice.

```
minibuffer:    Command #<DTP-CLOSURE 34465726> installed on Hyper-T.
```

```
you type:      m-X Deinstall Macro
minibuffer:    Key to deinstall:
you type:      h-T
```

The menu appears and asks you to specify in which of the three comtabs to deinstall the macro. Click on your choice.

```
minibuffer:    Command NIL installed on Hyper-T.
```

---

### *See...*

See "Key Bindings", page 198, for information about saving keyboard macros permanently.

---

### **More Features of the Keyboard Macro Facility**

The keyboard macro facility implemented with the `c-m-FUNCTION` key provides more features, such as an easy way to make tables.

#### `c-m-FUNCTION`

Reads a keyboard macro command, consisting of an optional numeric argument made up of any number of digits (0-9) followed by a non-numeric character, usually a letter. Each keyboard macro command must be preceded by the `c-m-FUNCTION` prefix. After typing the prefix, you may type `HELP` for a list of available keyboard macro commands.

**Creating New Commands: Keyboard Macros.** *cont'd.*

---

*Keyboard Macro Commands for c-m-FUNCTION*

- 0-9    Optional numeric argument.
- C     Calls a macro by name. Prompts in the minibuffer for the name of the macro.
- P     Begins a macro definition (same as c-X ( — see "Start Keyboard Macro").
- R     Ends a macro definition (same as c-X ) — see "End Keyboard Macro").
- M     Defines a named macro. Prompts for the name of the macro to define and then enters macro definition mode.
- S     Stops (aborts) macro definition (also c-G).
- D     Defines a named macro but does not execute it while reading its characters.
- SPACE Inserts pauses for user interaction in the macro (same as c-X Q — see "Defining an Interactive Keyboard Macro").
- A     Steps though characters on successive iterations (for example, letters and numbers). Asks for starting character, amount to increase (or decrease if negative) on each iteration.
- U     Allows typein terminated by c-m-FUNCTION R (MACRO R on LM-2). This allows you to stop while in the middle of defining the macro, do other things in the editor, and then go back and finish defining the macro.
- T     Allows typein every iteration.

The difference between c-m-FUNCTION U and c-m-FUNCTION T is that c-m-FUNCTION U allows typein while defining a macro that does not get stored in the macro, hence does not executed on subsequent iteration nor when the macro is called again. c-m-FUNCTION T allows typein on every iteration. As with c-m-FUNCTION U, the typein while defining the macro does not get stored in the macro. But on each subsequent iteration, new typein will be requested.

---

## Creating New Commands: Keyboard Macros, *cont'd.*

---

### *Example*

The following example shows how to create a macro that constructs a table using `c-m-FUNCTION A`.

```
you type: c-X (
Minibuffer: Macro-level: 1 *
you type: c-m-FUNCTION A
Minibuffer: Initial character (type a one-character string):
you type: a RETURN
Minibuffer: Amount by which to increase it (type a decimal number):
you type: 1 RETURN
                                (Zmacs inserts the a into the buffer.)
you type: c-2 c-6 c-X )
```

As you close the macro, Zmacs inserts into the buffer:

```
a b c d e f g h i j k l m n o p r s t u v w x y z
```

by executing the macro 26 times, increasing the letter once each time.

---

### *Example*

The following example shows how to create a macro that constructs a table using `c-m-FUNCTION A`, and this time, `c-m-FUNCTION T`, which allows `typein` during every iteration of the macro:

```
you type: c-X (
Minibuffer: Macro-level: 1 *
you type: Item SPACE
you type: c-m-FUNCTION A
Minibuffer: Initial character (type a one-character string):
you type: 1
Minibuffer: Amount by which to increase it (type a decimal number):
you type: 1
you type: TAB
you type: c-m-FUNCTION T
Minibuffer: Macro-level: 2 *
you type: Rosemary
you type: c-m-FUNCTION R
Minibuffer: Macro-level: 1 *
you type: RETURN
you type: c-5 c-X )
you type: Sage
you type: c-m-FUNCTION R
you type: Thyme
you type: c-m-FUNCTION R
you type: Parsley
you type: c-m-FUNCTION R
you type: Pepper
you type: c-m-FUNCTION R
```

**Creating New Commands: Keyboard Macros.** *cont'd.*

---

The table looks like this:

Item 1	Rosemary
Item 2	Sage
Item 3	Thyme
Item 4	Parsley
Item 5	Pepper

---

## Key Bindings

---

### Definition

A *key binding* is the set of specific keystrokes that invoke a specific command.

---

### How It Works: The Comtab

A *command table*, or *comtab*, assigns a command to each possible keystroke. While Zmacs is running, there is always a unique *selected comtab*, in which Zmacs finds the command that corresponds to each user keystroke. When you type a keystroke, Zmacs looks up the keystroke in the currently selected comtab, finds the appropriate command, and runs it. Usually the command's side-effects all occur within the buffer: Point might be moved and text might be deleted, inserted, or rearranged. Sometimes a command has more extensive side-effects. A command can alter or replace the selected comtab itself, in which case Zmacs looks up the next keystroke in the new command table.

Zmacs' *basic state* consists of the standard editor key bindings, which reside in one special command table, the *standard comtab* (*Zwei comtab*). The standard comtab interacts with the Zmacs comtab and the various mode-dependent comtabs. The typical selected comtab when in Zmacs is "unnamed" for mode-specific key bindings, which indirects to "Zmacs", which indirects to "Zwei". Although the standard comtab can be temporarily replaced, it is always reselected eventually, often after only one "nonstandard" keystroke.

A keystroke that functions as a prefix actually runs a command that replaces the standard comtab for one keystroke. This is the mechanism by which multikeystroke commands are implemented. For example, there are many two-stroke commands whose first keystroke is `c-x`. This keystroke runs a command that brings in its own comtab before interpreting the next stroke.

---

### Set Key

If you want to put a command on the keystroke of your choice, use `Set Key`. This command works for any of the already defined commands.

#### Set Key (*m-x*)

Installs a specified command on a specified key. If the key is currently holding a command prefix (like `c-x`), it asks you for another character so that you can redefine `c-x` commands. However, with a numeric argument, it assumes you want to redefine `c-x` itself and does not ask for another character.

---

**Key Bindings, cont'd.**

---

It assigns key bindings in the editor that are active in all buffers, and takes two arguments: the name of a command, and a keystroke to invoke it. It reads the name of the command in the minibuffer, completing any command name in any comtab.

---

**Install Command**

If you want to put a function on the keystroke of your choice, use Install Command. It takes a function, regards it as a command, and puts it on a key.

**Install Command (m-X)**

Installs a specified function as a command in the comtab, on a specified key. It takes two arguments: the name of the function (the current definition, that is, top level expression), and a keystroke to invoke it. (Zmacs treats as a definition any top-level expression having in functional position a symbol whose name begins "def".) If the key is currently holding a command prefix (like c-X), it asks you for another character so that you can redefine c-X commands. However, with a numeric argument, it assumes you want to redefine c-X itself and does not ask for another character.

---

*See...*

See "Key Bindings", page 198, for information about setting key bindings permanently.

---

---

## How to Specify Zmacs Variable Settings

---

### Definition

A *variable* is a name that is associated with a value, for example, a number or a string. Zmacs has editor variables that you can set for customization. (Variables can also be set automatically by major modes.)

You can distinguish the names of Zmacs variables from other Lisp variables by their names — the first letters are capitalized and the names contain spaces rather than hyphens. For example, Region Marking Mode is **zwei:\*region-marking-mode\*** internally.

---

### Finding Out About Zmacs Variables

To examine the value of a single Zmacs variable, use Describe Variable (m-X). To print a complete list of all variables, use List Variables (m-X).

---

### Describe Zmacs Variable

Describe Variable (m-X)

Displays the documentation and current value for a single Zmacs variable. It reads the variable name from the minibuffer, using completion.

---

### List Zmacs Variables

List Variables (m-X)

Lists *all* Zmacs variables and their values. With a numeric argument, this command also displays the documentation line for the variable.

---

### Variable Apropos

HELP V  
c-HELP V  
c-m-? V

Variable Apropos

Displays the names of all possible Zmacs variables containing a specific substring. With a numeric argument, this command also displays the documentation lines for the variables.

---

---

## How to Specify Zmacs Variable Settings. *cont'd.*

---

### *Example*

One example of such a Zmacs variable is the Fill Column variable, which specifies the width, in pixels, used in filling text.

For example, `c-1 HELP V` prompts in the minibuffer Variable Apropos (substring): and you type `fill col`. It does pattern matching on the variable name and thus matches Fill column, which displays: Fill column: 576. Width in pixels used in filling text.

---

### Set Variable

Set Variable (*m-X*)

Sets any existing Zmacs variable. This command reads the name of a variable (with completion), displays its current value and documentation, and prompts in the minibuffer for a new value. It does some checking to see that the new value has the right type.

Although either uppercase or lowercase works, you are encouraged to capitalize each word of the name for aesthetic reasons, since Zmacs stores the name as you give it.

---

### *Settable Zmacs Variables*

You can view all settable Zmacs variables with the List Variables command.

The following are some examples of variables that can be set with Set Variable. In addition, they can be set in init files by using the internal form of their names. For example, Region Marking Mode is `zwei:*region-marking-mode*` internally.

---

#### Region Marking Mode

Value: **:reverse-video** for setting the region to reverse video. The default is **:underline**.

#### Region Right Margin Mode

Value: **t**. Causes whatever marks the region (reverse video or underlining) to extend across unfilled space to the right margin. The default is **nil**.



## How to Specify Zmacs Variable Settings. *cont'd.*

---

### One Window Default

Controls which window remains selected after a One Window (c-x 1) command when you were using more than one window. Possible values:

**:current**  
**:other**  
**:top**  
**:bottom**

This feature operates best when the current layout has no more than two windows. The value **:current** is the only one that is always well defined with more than two windows on the screen.

### Check Unbalanced Parentheses When Saving

Controls whether Zmacs checks a file for unbalanced parentheses when you are saving the file. The check is on (t) by default. When it checks a file that you are saving and finds unbalanced parentheses, it queries you about whether to go ahead and save anyway. This applies to all major modes based on Lisp; it is ignored for text modes.

---

*See...*

See "Customizing the Editor in Init Files", page 196, for information about setting variables permanently.

---

## Customizing the Editor in Init Files

---

### Introduction

As you gain sophistication with the more advanced features, you will find the settings of parameters that most please you and put these into a command file (*init file*) that the system executes every time you log in.

---

### Creating An Init File

Create a file named *lisp-init.l* in your home directory on your host system and put your Zmacs customizations there.

This section contains examples of forms that you can place inside of a **login-forms** in your init file to customize the editor.

**login-forms** is a special form for wrapping around a set of forms in your init file. It evaluates the forms and arranges for them to be undone when you log out.

---

### Setting Editor Variables

The forms described show how to set Zmacs variables (the kind that Set Variable (M-X) sets).

---

#### *Ordering Buffer Lists*

```
(SETQ ZWEI:*SORT-ZMACS-BUFFER-LIST* NIL)
```

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*

```
(SETQ ZWEI:*DEFAULT-PACKAGE* NIL)
```

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*

```
(SETQ ZWEI:*DEFAULT-MAJOR-MODE* 'TEXT)
```

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.

---

---

## Customizing the Editor in Init Files, *cont'd.*

---

### *Setting Find File*

#### *Not To Create New Files*

```
(SETQ ZWEI:*FIND-FILE-NOT-FOUND-IS-AN-ERROR* T)
```

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.

---

### *Setting Goal*

#### *Column for Real Line Commands*

```
(SETQ ZWEI:*PERMANENT-REAL-LINE-GOAL-XPOS* 0)
```

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*

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

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

---

### **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*

```
(SETQ ZWEI:LISP-MODE-HOOK 'ZWEI:ELECTRIC-SHIFT-LOCK-IF-APPROPRIATE)
```

tells Lisp major mode to turn on Electric Shift Lock minor mode unless the buffer has a Lowercase attribute. The effect is that by default Lisp code is written in upper case.

---

---

## Customizing the Editor in Init Files, *cont'd.*

---

### *Auto Fill in Text Mode*

```
(SETQ ZWEI:TEXT-MODE-HOOK 'ZWEI:AUTO-FILL-IF-APPROPRIATE)
```

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.

---

### **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.

---

### *Balanced Quotation Marks and Asterisks*

```
ZWEI:(SET-COMTAB *STANDARD-COMTAB*
      '(#\m-/" COM-MAKE-/(/)
        #\c-m-/" COM-MOVE-OVER/)
        #\m-/* COM-MAKE-/(/)
        #\c-m-/* COM-MOVE-OVER-/)
      ))
```

defines commands to insert balanced pairs of quotation marks or asterisks into the buffer. For example, one can type an asterisked special variable name as `m-* FOO`, which inserts `*FOO*` into the buffer, ensuring that one does not forget to type the trailing asterisk.

---

### *White Space In Lisp Code*

```
ZWEI:(SET-COMTAB *STANDARD-CONTROL-X-COMTAB*
      '(#\SP COM-CANONICALIZE-WHITESPACE))
```

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.

---

---

## Customizing the Editor in Init Files, *cont'd.*

---

### *c-m-L on the SQUARE Key*

```
ZWEI:(SET-COMTAB *ZMACS-COMTAB*  
      '(\SQUARE COM-SELECT-PREVIOUS-BUFFER))
```

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-[L]*

```
ZWEI:(SET-COMTAB *ZMACS-COMTAB*  
      '(\m-MOUSE-L COM-EDIT-DEFINITION))
```

This makes clicking the Left mouse button while holding down the Meta key do what `m-.` does. Invoking this command from the mouse is convenient when you specify the name of the definition to be edited by pointing at it rather than typing it.

---



## Appendix A Help Command Summary

This section lists the names of the available help commands grouped according to the context in which they are available. The purpose of this section is to summarize the capabilities and to help you determine both the overall contexts for which you can find help and a particular function that might be what you are looking for.

### Zmacs commands

#### for finding out

#### about the state of buffers

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

### Zmacs commands

#### for finding out

#### about the state of Zmacs

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

### Zmacs commands

#### for finding out

#### about Lisp

- Brief Documentation (c-sh-D)
- Describe Variable At Point (c-sh-V)
- Edit Callers (m-X)
- Edit Definition (m-.)

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

**Zmacs commands  
for finding out  
about flavors**

Describe Flavor (m-X)  
Edit Combined Methods (m-X)  
Edit Methods (m-X)  
List Combined Methods (m-X)  
List Methods (m-X)

**Zmacs commands  
for interacting  
with Lisp**

Break (SUSPEND — BREAK on an LM-2)  
Compile And Exit (m-Z)  
Compile Buffer (m-X)  
Compile Changed Definitions (m-X)  
Compile Changed Definitions Of Buffer (m-sh-C, m-X)  
Compile File (m-X)  
Compile Region (c-sh-C, m-X)  
Compiler Warnings (m-X)  
Edit Compiler Warnings (m-X)  
Evaluate And Exit (c-m-Z)  
Evaluate And Replace Into Buffer (m-X)  
Evaluate Buffer (m-X)  
Evaluate Changed Definitions (m-X)  
Evaluate Changed Definitions Of Buffer (m-sh-E, m-X)  
Evaluate Into Buffer (m-X)  
Evaluate Minibuffer ((ESCAPE — m-ALTMODE on an LM-2))  
Evaluate Region (c-sh-E, m-X)  
Evaluate Region Hack (m-X)  
Evaluate Region Verbose (c-m-sh-E)  
Load Compiler Warnings (m-X)  
Macro Expand Expression (c-sh-M, m-X)  
Trace (m-X)  
Quit (c-Z)



## Index

<b>!</b>	<b>!</b>	<b>!</b>
	! Dired command 140	
Exclamation point	(!) line continuation indicator 20, 46	
<b>\$</b>	<b>\$</b>	<b>\$</b>
	\$ Dired command 139	
<b>)</b>	<b>)</b>	<b>)</b>
Move Over	) 56	
<b>*</b>	<b>*</b>	<b>*</b>
	* buffer status indicator 19	
<b>+</b>	<b>+</b>	<b>+</b>
	+ flag in Zmacs 103	
<b>,</b>	<b>,</b>	<b>,</b>
	, Dired command 136	
<b>1</b>	<b>1</b>	<b>1</b>
	c-X 1 Zmacs command 118	
<b>2</b>	<b>2</b>	<b>2</b>
	c-X 2 Zmacs command 117	
<b>3</b>	<b>3</b>	<b>3</b>
	c-X 3 Zmacs command 117	
<b>4</b>	<b>4</b>	<b>4</b>
	c-X 4 Zmacs command 117	
<b>8</b>	<b>8</b>	<b>8</b>
	c-X 8 Zmacs command 117	

;	;	;
c-X	; Zmacs command	158
Semicolon	(;) comment indicator	157
=	=	=
c-X	= Dired command	137
	= Zmacs command	37
?	?	?
HELP	? Dired command	135
	? Zmacs command	12
A	A	A
c-X	A Dired command	141
HELP	A Zmacs command	112
Word	A Zmacs command	12, 35
Word	Abbrev mode	18
Dired	Abbrev Mode (m-X) Zmacs command	181
	Abort	135
	Abort At Top Level	32
	ABORT Dired command	135
	Abort Patch (m-X) Zmacs command	174
	ABORT Zmacs command	32
Warnings	about file attribute lists	126
Zmacs commands for finding out	about flavors	202
Zmacs commands for finding out	about Lisp	201
Zmacs commands for finding out	about the state of buffers	201
Zmacs commands for finding out	about the state of Zmacs	201
Finding Out	About Zmacs Commands	34
	Accidental deletion	33
	Add Patch (m-X) Zmacs command	173
	Add Patch Changed Definitions (m-X) Zmacs command	173
	Add Patch Changed Definitions of Buffer (m-X) Zmacs command	173
	Additional Notation Conventions	7
	Adjust paragraph	83
	Adjust region	83
Macro Expand Expression	All	165
Select	All Buffers As Tag Table (m-X) Zmacs command	94
Undo	all changes to buffer	109
Save	All Files (m-X) Zmacs command	108
Point motion	along nesting level	54
Fast Where	Am I	37
Where	Am I	37
Point Motion	Among Top-Level Expressions	55
	Any Extended Command	6
	Append Next Kill	65
	Append To Buffer	112
	Append To File (m-X) Zmacs command	112
	Appending, Prepending, and Inserting Text	112
Dired	Apply Function	141
	Applying Functions to Files	141
Searching for	Appropriate Commands	35

Completion Variable Variable Echo  
 Quadruple Numeric Quick  
 Display Echoing Negative Numeric Numeric  
 Positioning Window

Apropos 12  
 Apropos 44  
 Apropos 193  
 Apropos Zmacs command 193  
 Area 17  
 Arg 22  
 Arglist 37  
 Arglist (m-X) Zmacs command 37  
 argument list 37  
 arguments 22  
 Arguments 51  
 arguments 20, 22, 34, 51  
 Around Definition 48  
 Assign key bindings 192  
 Association of buffers with files 27  
 Association of files with buffers 27  
 Atom Query Replace (m-X) Zmacs command 93  
 Atom Word Mode (m-X) Zmacs command 180

Backspace file attribute 128  
 Base file attribute 128  
 Lowercase file attribute 128  
 Nofill file attribute 129  
 Patch-File file attribute 129  
 Tab-Width file attribute 130  
 Unknown attribute 126  
 Vsp file attribute 130

File Attribute Checking 126  
 Attribute list 125  
 Describe Attribute List 136  
 Dired Describe Attribute List 136  
 Reparse Attribute List (m-X) Zmacs command 127  
 Reparse Attribute List (m-X) Zmacs command 125  
 Update Attribute List (m-X) Zmacs command 125  
 Attribute lists 126

Warnings about file attribute lists 126  
 Buffer attributes 125  
 Buffer and File Attributes 125  
 File attributes 125  
 Set commands for file and buffer attributes 127

Auto Fill Lisp Comments Mode (c-m-X) Zmacs command 180  
 Auto Fill mode 18, 129, 179  
 Auto Fill Mode (m-X) Zmacs command 180  
 Dired Automatic 138, 139  
 Locating and Replacing Strings Automatically 90

**B**

**B**  
 c-X B Zmacs command 27, 105  
 Back To Indentation 151  
 Set Backspace (m-X) Zmacs command 128  
 Backspace file attribute 128  
 File backup flag 140  
 Backward 23, 52  
 Backward character 51  
 Backward Kill Sentence 25, 72  
 Backward Kill Sexp 25, 69  
 Backward Kill Word 25, 68

**B**

	Backward List	53, 54
	Backward Paragraph	23, 58
	Backward Sentence	23, 53
	Backward Sexp	54
Erase	backward to start of line	71
	Backward Up List	55
Kill	Backward Up List (c-m-X) Zmacs command	69
	Backward Word	23, 52
	Base	126
Set	Base (m-X) Zmacs command	128
	Base file attribute	128
	Beep	32, 33
Goto	Beginning	24, 60
Mark	Beginning	81
	Beginning Of Definition	55
	Beginning of Line	23, 57
	beginning of line	48
Move cursor to	Binding (m-X) Zmacs command	92
Query Replace Let	bindings	192
Assign key	bindings	12
Command key	bindings	36
Extended command key	Bindings	191
Key	Blank Lines	26, 153
Delete	Bolio mode	145
	Breaking a line	20
	Brief Documentation	37
	Buffer	112
Append To	Buffer	27
Creating a	buffer	109
Current	buffer	60
Cursor motion within	Buffer	108
Decrypt	Buffer	108
Encrypt	Buffer	108
*Function-Specs-to-Edit-m*	buffer	11
Insert text from register into	buffer	78
Kill	Buffer	110
Mark	buffer	81
Motion with Respect to the Whole	Buffer	60
Moving to end of	buffer	60
Select	Buffer	27, 105
Select Default Previous	Buffer	105
Select Previous	Buffer	105
Selected	buffer	104
Undo all changes to	buffer	109
View	Buffer	107
Add Patch Changed Definitions of	Buffer (m-X) Zmacs command	173
Compile	Buffer (m-X) Zmacs command	162
Compile Changed Definitions of	Buffer (m-X) Zmacs command	162
Decrypt	Buffer (m-X) Zmacs command	108
Edit Changed Definitions of	Buffer (m-X) Zmacs command	168
Encrypt	Buffer (m-X) Zmacs command	108
Evaluate	Buffer (m-X) Zmacs command	160
Evaluate and Replace Into	Buffer (m-X) Zmacs command	92, 161
Evaluate Changed Definitions of	Buffer (m-X) Zmacs command	161
Evaluate Into	Buffer (m-X) Zmacs command	160
Hardcopy	Buffer (m-X) Zmacs command	108
Insert	Buffer (m-X) Zmacs command	112
List Changed Definitions of	Buffer (m-X) Zmacs command	169
Multiple Query Replace From	Buffer (m-X) Zmacs command	92

Rename	Buffer (m-X) Zmacs command	108
Revert	Buffer (m-X) Zmacs command	109
View	Buffer (m-X) Zmacs command	107
	Buffer and File Attributes	125
	Buffer attributes	125
Set commands for file and	buffer attributes	127
	Buffer Commands	105
	Buffer History	104
Current	buffer information	17
	Buffer names	18
	Buffer pointers	74
*	buffer status indicator	19
	Buffers	15, 18, 27
Association of files with	buffers	27
Comparing Files and	Buffers	113
List	Buffers	106
Multiple	buffers	102
Possibility	Buffers	98
Selecting, Listing, and Examining	Buffers	104
Support	buffers	98
Zmacs commands for finding out about the state of	buffers	201
Edit	Buffers (m-X) Zmacs command	107
Kill Or Save	Buffers (m-X) Zmacs command	110
Kill Some	Buffers (m-X) Zmacs command	110
Creating and Saving	Buffers and Files	27
Manipulating	Buffers and Files	101
Working With	Buffers and Files	102
Select All	Buffers As Tag Table (m-X) Zmacs command	94
Association of	buffers with files	27
List	Buffers Zmacs command	103

C

C

C

SELECT	C	30
	C Dired command	137
HELP	C Zmacs command	12, 34
	c-% Zmacs command	90
	c-. Zmacs command	98, 168
	c-/ completion command	13, 44
	c-@ c-m-Y yank command	14
	c-; Zmacs command	157
c-X	c-; Zmacs command	159
	c-< Zmacs command	81
	c-= Zmacs command	37
	c-> Zmacs command	81
	c-? completion command	13, 44
	c-A Zmacs command	23, 57
	c-B Zmacs command	23, 52
c-X	c-B Zmacs command	103, 106
	c-D Dired command	138
	c-D Zmacs command	25, 33, 66
c-X	c-D Zmacs command	119
	c-E Zmacs command	23, 57
	c-F Zmacs command	23, 51
c-X	c-F Zmacs command	28, 29, 108
	c-G Zmacs command	32, 33
c-X	c-G Zmacs command	32
	c-HELP V Zmacs command	193

c-X	c-I Zmacs command	152
	c-K Dired command	138
	c-K Zmacs command	25, 71
	c-L Zmacs command	47
c-X	c-L Zmacs command	148
	c-m-( Zmacs command	55
	c-m-) Zmacs command	55
	c-m-; Zmacs command	157
	c-m-? Y Zmacs command	193
	c-m-@ Zmacs command	80
	c-m-A Zmacs command	55
	c-m-B Zmacs command	54
	c-m-D Zmacs command	54
	c-m-E Zmacs command	56
	c-m-F Zmacs command	54
	c-m-H Zmacs command	80
	c-m-K Zmacs command	25, 69
	c-m-L Zmacs command	105
c-X	c-m-L Zmacs command	105
	c-m-M Zmacs command	151
	c-m-N Zmacs command	53
	c-m-O Zmacs command	153
	c-m-P Zmacs command	53
	c-m-Q Zmacs command	151
	c-m-R Zmacs command	48
	c-m-RETURN Zmacs command	151
	c-m-RUBOUT Zmacs command	25, 69
	c-m-sh-E Zmacs command	160
	c-m-SPACE Zmacs command	76
c-X	c-m-SPACE Zmacs command	76
	c-m-T Zmacs command	70
	c-m-U Zmacs command	55
	c-m-V Zmacs command	118
	(c-m-X) Zmacs command	180
	(c-m-X) Zmacs command	69
	c-m-Y yank command	14
c-Ø	c-m-Y yank command	14
	c-m-Z Zmacs command	161
	c-m-[ Zmacs command	55
	c-m-\ Zmacs command	151
	c-m-] Zmacs command	56
	c-m-^ Zmacs command	152
	c-N Dired command	136
	c-N Zmacs command	23, 56, 183
c-X	c-N Zmacs command	57
	c-O c-Y yank command	13
	c-O Zmacs command	153
c-X	c-O Zmacs command	26, 153
	c-P Dired command	136
	c-P Zmacs command	23, 56
c-X	c-P Zmacs command	81
	c-R Zmacs command	87
c-X	c-S Zmacs command	28, 109
	c-sh-A Zmacs command	37
	c-sh-C Zmacs command	82
	c-sh-D Zmacs command	37
	c-sh-E Zmacs command	160
	c-sh-M Zmacs command	165

Auto Fill Lisp Comments Mode  
Kill Backward Up List

	c-sh-V Zmacs command	38
	c-SPACE Zmacs command	76
	c-T Zmacs command	66
c-X	c-T Zmacs command	71
	c-U Zmacs command	22
c-X	c-U Zmacs command	148
	c-V Zmacs command	24, 47
c-X	c-V Zmacs command	109
	c-W Zmacs command	25, 82
c-X	c-W Zmacs command	29, 109
	c-X 1 Zmacs command	118
	c-X 2 Zmacs command	117
	c-X 3 Zmacs command	117
	c-X 4 Zmacs command	117
	c-X 8 Zmacs command	117
	c-X ; Zmacs command	158
	c-X = Zmacs command	37
	c-X A Zmacs command	112
	c-X B Zmacs command	27, 105
	c-X c-; Zmacs command	159
	c-X c-B Zmacs command	103, 106
	c-X c-D Zmacs command	119
	c-X c-F Zmacs command	28, 29, 108
	c-X c-G Zmacs command	32
	c-X c-I Zmacs command	152
	c-X c-L Zmacs command	148
	c-X c-m-L Zmacs command	105
	c-X c-m-SPACE Zmacs command	76
	c-X c-N Zmacs command	57
	c-X c-O Zmacs command	26, 153
	c-X c-P Zmacs command	81
	c-X c-S Zmacs command	28, 109
	c-X c-T Zmacs command	71
	c-X c-U Zmacs command	148
	c-X c-V Zmacs command	109
	c-X c-W Zmacs command	29, 109
	c-X c-X Zmacs command	77
	c-X D Zmacs command	131
	c-X E Zmacs command	184
	c-X F Zmacs command	27, 180
	c-X H Zmacs command	81
	c-X K Zmacs command	110
	c-X L Zmacs command	38
	c-X O Zmacs command	118
	c-X Q Zmacs command	185
	c-X RUBOUT Zmacs command	25, 72
	c-X S Zmacs command	27
	c-X T Zmacs command	82
	c-X V Zmacs command	107
	c-X W Zmacs command	27
	c-X Zmacs command	86, 183
c-X	c-X Zmacs command	77
	c-X ^ Zmacs command	117
	c-Y yank command	13
c-O	c-Y yank command	13
	c-Z Zmacs command	30
	Call Last Kbd Macro	184
Using the mouse to	call up System menu	30

Edit	Callers (m-X) Zmacs command	169
List	Callers (m-X) Zmacs command	170
Multiple Edit	Callers (m-X) Zmacs command	170
Multiple List	Callers (m-X) Zmacs command	171
	Cancel last command	32
	Cancel response	32
	Canonical types	122
	Carriage return	5
Changing	Case	148
Changing	Case and Indentation	147
Changing	Case of Region	148
	Center Line	150
Dired	Change File Properties	136
	Change File Properties (m-X) Zmacs command	123
Add Patch	Changed Definitions (m-X) Zmacs command	173
Compile	Changed Definitions (m-X) Zmacs command	162
Edit	Changed Definitions (m-X) Zmacs command	167
Evaluate	Changed Definitions (m-X) Zmacs command	161
Add Patch	Changed Definitions of Buffer (m-X) Zmacs command	173
Compile	Changed Definitions of Buffer (m-X) Zmacs command	162
Edit	Changed Definitions of Buffer (m-X) Zmacs command	168
Evaluate	Changed Definitions of Buffer (m-X) Zmacs command	161
List	Changed Definitions of Buffer (m-X) Zmacs command	169
Undo all	changes to buffer	109
	Changing Case	148
	Changing Case and Indentation	147
	Changing Case of Region	148
Backward	character	51
Forward	character	51
Motion by	character	51
RUBOUT Zmacs	character	66
Prefix	character commands	5
Deleting and Transposing	Characters	66
Exchange	Characters	66
Inserting	Characters	20
Newline	characters	20
File Attribute	Check Unbalanced Parentheses When Saving	195
	Checking	126
	Clean Directory (m-X) Zmacs command	123
	Clear	71
	CLEAR-INPUT Zmacs command	71
L2:Move to point mouse	click	49
L:Move point mouse	click	49
M2:Save/Kill/Yank mouse	click	49
M:Mark thing mouse	click	49
R2:System menu mouse	click	50
R:Menu mouse	click	50
Current meaning of mouse	clicks	49
Commenting	Code	157
Finding source	code	156
Lowercase	Code in Region	148
Uppercase	Code in Region (m-X) Zmacs command	148
Editing the source	code of a function	11



Locating Source	Code to Edit	166
Set Comment	Column	158
Set Fill	Column	180
Set Goal	Column	57
Default	column position	57
! Dired	command	140
* Dired	command	139
, Dired	command	136
= Dired	command	137
? Dired	command	135
A Dired	command	141
ABORT Dired	command	135
Abort Patch (m-X) Zmacs	command	174
ABORT Zmacs	command	32
Add Patch (m-X) Zmacs	command	173
Add Patch Changed Definitions (m-X) Zmacs	command	173
	Add Patch Changed Definitions of Buffer (m-X) Zmacs	command 173
Any Extended	Command	6
Append To File (m-X) Zmacs	command	112
Arglist (m-X) Zmacs	command	37
Atom Query Replace (m-X) Zmacs	command	93
Atom Word Mode (m-X) Zmacs	command	180
Auto Fill Lisp Comments Mode (c-m-X) Zmacs	command	180
Auto Fill Mode (m-X) Zmacs	command	180
C Dired	command	137
c-% Zmacs	command	90
c-. Zmacs	command	98, 168
c-/ completion	command	13, 44
c-Ø c-m-Y yank	command	14
c-; Zmacs	command	157
c-< Zmacs	command	81
c-= Zmacs	command	37
c-> Zmacs	command	81
c-? completion	command	13, 44
c-A Zmacs	command	23, 57
c-B Zmacs	command	23, 52
c-D Dired	command	138
c-D Zmacs	command	25, 33, 66
c-E Zmacs	command	23, 57
c-F Zmacs	command	23, 51
c-G Zmacs	command	32, 33
c-HELP V Zmacs	command	193
c-K Dired	command	138
c-K Zmacs	command	25, 71
c-L Zmacs	command	47
c-m-{ Zmacs	command	55
c-m-) Zmacs	command	55
c-m-; Zmacs	command	157
c-m-? V Zmacs	command	193
c-m-@ Zmacs	command	80
c-m-A Zmacs	command	55
c-m-B Zmacs	command	54
c-m-D Zmacs	command	54
c-m-E Zmacs	command	56
c-m-F Zmacs	command	54
c-m-H Zmacs	command	80
c-m-K Zmacs	command	25, 69

c-m-L Zmacs	command	105
c-m-M Zmacs	command	151
c-m-N Zmacs	command	53
c-m-O Zmacs	command	153
c-m-P Zmacs	command	53
c-m-Q Zmacs	command	151
c-m-R Zmacs	command	48
c-m-RETURN Zmacs	command	151
c-m-RUBOUT Zmacs	command	25, 69
c-m-sh-E Zmacs	command	160
c-m-SPACE Zmacs	command	76
c-m-T Zmacs	command	70
c-m-U Zmacs	command	55
c-m-V Zmacs	command	118
c-m-Y yank	command	14
c-m-Z Zmacs	command	161
c-m-[ Zmacs	command	55
c-m-\ Zmacs	command	151
c-m-] Zmacs	command	56
c-m-^ Zmacs	command	152
c-N Dired	command	136
c-N Zmacs	command	23, 56, 183
c-O c-Y yank	command	13
c-O Zmacs	command	153
c-P Dired	command	136
c-P Zmacs	command	23, 56
c-R Zmacs	command	87
c-sh-A Zmacs	command	37
c-sh-C Zmacs	command	82
c-sh-D Zmacs	command	37
c-sh-E Zmacs	command	160
c-sh-M Zmacs	command	165
c-sh-V Zmacs	command	38
c-SPACE Zmacs	command	76
c-T Zmacs	command	66
c-U Zmacs	command	22
c-V Zmacs	command	24, 47
c-W Zmacs	command	25, 82
c-X 1 Zmacs	command	118
c-X 2 Zmacs	command	117
c-X 3 Zmacs	command	117
c-X 4 Zmacs	command	117
c-X 8 Zmacs	command	117
c-X ; Zmacs	command	158
c-X = Zmacs	command	37
c-X A Zmacs	command	112
c-X B Zmacs	command	27, 105
c-X c-; Zmacs	command	159
c-X c-B Zmacs	command	103, 106
c-X c-D Zmacs	command	119
c-X c-F Zmacs	command	28, 29, 108
c-X c-G Zmacs	command	32
c-X c-I Zmacs	command	152
c-X c-L Zmacs	command	148
c-X c-m-L Zmacs	command	105
c-X c-m-SPACE Zmacs	command	76
c-X c-N Zmacs	command	57
c-X c-O Zmacs	command	26, 153

c-X c-P Zmacs	command	81
c-X c-S Zmacs	command	28, 109
c-X c-T Zmacs	command	71
c-X c-U Zmacs	command	148
c-X c-V Zmacs	command	109
c-X c-W Zmacs	command	29, 109
c-X c-X Zmacs	command	77
c-X D Zmacs	command	131
c-X E Zmacs	command	184
c-X F Zmacs	command	27, 180
c-X H Zmacs	command	81
c-X K Zmacs	command	110
c-X L Zmacs	command	38
c-X O Zmacs	command	118
c-X Q Zmacs	command	185
c-X RUBOUT Zmacs	command	25, 72
c-X S Zmacs	command	27
c-X T Zmacs	command	82
c-X V Zmacs	command	107
c-X W Zmacs	command	27
c-X Zmacs	command	86, 183
c-X ^ Zmacs	command	117
c-Y yank	command	13
c-Z Zmacs	command	30
Cancel last	command	32
Change File Properties (m-X) Zmacs	command	123
Clean Directory (m-X) Zmacs	command	123
CLEAR-INPUT Zmacs	command	71
Compile Buffer (m-X) Zmacs	command	162
Compile Changed Definitions (m-X) Zmacs	command	162
Compile Changed Definitions of Buffer (m-X) Zmacs	command	162
Compile File (m-X) Zmacs	command	162
Compile Region (m-X) Zmacs	command	82
Compiler Warnings (m-X) Zmacs	command	163
COMPLETE completion	command	13, 42
Copy File (m-X) Zmacs	command	121
Create Directory (m-X) Zmacs	command	123
Create Link (m-X) Zmacs	command	122
D Dired	command	138
Decrypt Buffer (m-X) Zmacs	command	108
Deinstall Macro (m-X) Zmacs	command	186
Delete File (m-X) Zmacs	command	122
Describe Variable (m-X) Zmacs	command	193
Dired (m-X) Zmacs	command	131
E Dired	command	137
Edit Buffers (m-X) Zmacs	command	107
Edit Callers (m-X) Zmacs	command	169
Edit Changed Definitions (m-X) Zmacs	command	167
Edit Changed Definitions of Buffer (m-X) Zmacs	command	168
Edit Compiler Warnings (m-X) Zmacs	command	163
Edit Definition	Command	166
Edit File Warnings (m-X) Zmacs	command	163
Electric Font Lock Mode (m-X) Zmacs	command	180
Electric Shift Lock Mode (m-X) Zmacs	command	180
EMACS Mode (m-X) Zmacs	command	181
Encrypt Buffer (m-X) Zmacs	command	108
END completion	command	13, 43
END Dired	command	134

End Kbd Macro Zmacs	command	183
Evaluate and Replace Into Buffer (m-X) Zmacs	command	92, 161
Evaluate Buffer (m-X) Zmacs	command	160
Evaluate Changed Definitions (m-X) Zmacs	command	161
Evaluate Changed Definitions of Buffer (m-X) Zmacs	command	161
Evaluate Into Buffer (m-X) Zmacs	command	160
Evaluate Region (m-X) Zmacs	command	160, 161
Extended	Command	5
Find File In Fundamental Mode (m-X) Zmacs	command	110
Find File Zmacs	command	28, 29
Find Files in Tag Table (m-X) Zmacs	command	97
Find Unbalanced Parentheses (m-X) Zmacs	command	38
Finding the right	command	35
Finish Patch (m-X) Zmacs	command	174
H Dired	command	138
Hardcopy Buffer (m-X) Zmacs	command	108
Hardcopy File (m-X) Zmacs	command	120
HELP ? Zmacs	command	12
HELP A Zmacs	command	12, 35
HELP C Zmacs	command	12, 34
HELP completion	command	13
HELP D Zmacs	command	12, 35
HELP Dired	command	135
HELP L Zmacs	command	12, 36
HELP SPACE Zmacs	command	12
HELP U Zmacs	command	12, 36
HELP V Zmacs	command	12, 36, 193
HELP W Zmacs	command	12
Insert Buffer (m-X) Zmacs	command	112
Insert File (m-X) Zmacs	command	112
Install Command (m-X) Zmacs	command	192
Install Macro (m-X) Zmacs	command	186
K Dired	command	138
Kill Backward Up List (c-m-X) Zmacs	command	69
Kill Or Save Buffers (m-X) Zmacs	command	110
Kill Some Buffers (m-X) Zmacs	command	110
LINE Zmacs	command	151
Lisp Mode (m-X) Zmacs	command	156
List Buffers Zmacs	command	103
List Callers (m-X) Zmacs	command	170
List Changed Definitions of Buffer (m-X) Zmacs	command	169
List Definitions (m-X) Zmacs	command	168
List Files (m-X) Zmacs	command	119
List Tag Tables (m-X) Zmacs	command	95
List Variables (m-X) Zmacs	command	193
Load Compiler Warnings (m-X) Zmacs	command	163
m- <del>X</del> Zmacs	command	90
m-( Zmacs	command	164
m-) Zmacs	command	56
m-. Zmacs	command	166
m-; Zmacs	command	157
m-< Zmacs	command	24, 60
m-= Zmacs	command	38
m-> Zmacs	command	24, 60
m-@ Zmacs	command	80
m-A Zmacs	command	23, 53
m-B Zmacs	command	23, 52
m-C Zmacs	command	148

m-D Zmacs	command	25, 68
m-E Zmacs	command	53
m-ESCAPE Zmacs	command	160
m-F Zmacs	command	23, 52
m-H Zmacs	command	80
m-K Zmacs	command	25, 72
m-L Zmacs	command	148
m-LINE Zmacs	command	158
m-M Zmacs	command	151
m-N Zmacs	command	158
m-O Zmacs	command	152
m-P Zmacs	command	158
m-R Zmacs	command	48
m-RETURN Zmacs	command	151
m-RUBOUT Zmacs	command	25, 68
m-S Zmacs	command	150
m-sh-C Zmacs	command	162
m-sh-D Zmacs	command	37
m-sh-E Zmacs	command	161
m-T Zmacs	command	68
m-U Zmacs	command	148
m-V Zmacs	command	24, 48
m-W Zmacs	command	82
m-Y yank	command	13, 14, 64
m-Z Zmacs	command	162
m-[ Zmacs	command	23, 58
m-\ Zmacs	command	26
m-] Zmacs	command	23, 58
m-^ Zmacs	command	26, 152
Macro Expand Expression (m-X) Zmacs	command	165
Multiple Edit Callers (m-X) Zmacs	command	170
Multiple List Callers (m-X) Zmacs	command	171
Multiple Query Replace (m-X) Zmacs	command	91
Multiple Query Replace From Buffer (m-X) Zmacs	command	92
N Dired	command	138
Name Last Kbd Macro (m-X) Zmacs	command	185
Overwrite Mode (m-X) Zmacs	command	181
P Dired	command	140
Prepend To File (m-X) Zmacs	command	112
Q Dired	command	134
Query Replace (m-X) Zmacs	command	90
Query Replace Last Kill (m-X) Zmacs	command	92
Query Replace Let Binding (m-X) Zmacs	command	92
R Dired	command	137
Reap File (m-X) Zmacs	command	122
Rename Buffer (m-X) Zmacs	command	108
Rename File (m-X) Zmacs	command	120
Reparsing Attribute List (m-X) Zmacs	command	125
Repeat Last Minibuffer	Command	41, 64
Replace String (m-X) Zmacs	command	90
RETURN completion	command	13
RETURN Zmacs	command	43
Revert Buffer (m-X) Zmacs	command	109
RUBOUT Dired	command	138
RUBOUT Zmacs	command	25, 33
Save All Files (m-X) Zmacs	command	108
Save File Zmacs	command	28
Select All Buffers As Tag Table (m-X) Zmacs	command	94

Select System As Tag Table (m-X) Zmacs	command	95
Select Tag Table (m-X) Zmacs	command	94
Set Backspace (m-X) Zmacs	command	128
Set Base (m-X) Zmacs	command	128
Set Fonts (m-X) Zmacs	command	128
Set Key (m-X) Zmacs	command	191
Set Lowercase (m-X) Zmacs	command	128
Set Nofill (m-X) Zmacs	command	128
Set Package (m-X) Zmacs	command	126, 127, 128
Set Patch File (m-X) Zmacs	command	128
Set Tab Width (m-X) Zmacs	command	128
Set Variable (m-X) Zmacs	command	194
Set Visited File Name (m-X) Zmacs	command	110
Set Vsp (m-X) Zmacs	command	128
Sort Lines (m-X) Zmacs	command	100
Sort Paragraphs (m-X) Zmacs	command	100
Sort Via Keyboard Macros (m-X) Zmacs	command	100
Source Compare (m-X) Zmacs	command	113
Source Compare Merge (m-X) Zmacs	command	113
SPACE completion	command	13
SPACE Dired	command	136
SPACE Zmacs	command	43
Start Patch (m-X) Zmacs	command	172
Start Private Patch (m-X) Zmacs	command	172
TAB Zmacs	command	150
Tags Multiple Query Replace (m-X) Zmacs	command	96
Tags Query Replace (m-X) Zmacs	command	95
Tags Search (m-X) Zmacs	command	95
Trace (m-X) Zmacs	command	38
U Dired	command	138
Uncomment Region (m-X) Zmacs	command	159
Update Attribute List (m-X) Zmacs	command	125
Uppercase Code in Region (m-X) Zmacs	command	148
V Dired	command	137
Variable Apropos Zmacs	command	193
View Buffer (m-X) Zmacs	command	107
View File (m-X) Zmacs	command	120
View File Properties (m-X) Zmacs	command	120
View Kbd Macro (m-X) Zmacs	command	184
Visit Tag Table (m-X) Zmacs	command	97
Word Abbrev Mode (m-X) Zmacs	command	181
Write File Zmacs	command	29
Install	Command (m-X) Zmacs command	192
Command completion		42
Command completion		43
Command description		12
Command history		33
Command history		14
Command key bindings		12
Command key bindings		36
Command name completion		44
Command Names		4
Command table		191
Command tables		5
Commands		4
Buffer	Commands	105
Comment	commands	157
Compilation	commands	162
Impossible-is-OK		
Yanking in the		
Extended		

Completion	Commands	42
Cursor movement	commands	23, 51
Delete	commands	25
Dired	Commands	132
Editor menu	commands	39
Evaluation	commands	160
Extended	commands	4, 5, 35
File Manipulation	Commands	119
Finding Out About Zmacs	Commands	34
Kill	commands	25
List the last sixty	commands	36
Local names of	commands	6
Motion	Commands	51
Mouse-sensitive Zmacs	commands	50
Names of	commands	5, 34
Online documentation for	commands	34
Prefix	Commands	35
Prefix character	commands	5
Region-Manipulating	Commands	82
Searching for Appropriate	Commands	35
Set	commands	128
Window	Commands	117
Set	commands for file and buffer attributes	127
Zmacs	commands for finding out about flavors	202
Zmacs	commands for finding out about Lisp	201
Zmacs	commands for finding out about the state of buffers	201
Zmacs	commands for finding out about the state of Zmacs	201
Zmacs	commands for interacting with Lisp	202
Commands for manipulating files		119
Commands to Mark Regions		80
Creating New	Commands: Keyboard Macros	183
Indent For	Comment	157
Kill	Comment	157
Set	Comment Column	158
Semicolon (;)	Comment commands	157
Down	comment indicator	157
Indent New	Comment Line	158
Up	Comment Line	158
Comment Out Region		159
Commenting Code		157
Commenting Regions		159
Auto Fill Lisp	Comments Mode (c-m-X) Zmacs command	180
Source	Compare	119
Source	Compare (m-X) Zmacs command	113
Source	Compare Merge	114
Source	Compare Merge (m-X) Zmacs command	113
Comparing file versions		137
Comparing Files and Buffers		113
Evaluation and	Compilation	160
Compilation commands		162
Compile and Exit		162
Compile Buffer (m-X) Zmacs command		162
Compile Changed Definitions (m-X) Zmacs command		162
Compile Changed Definitions of Buffer (m-X) Zmacs		

command 162  
 Compile File (m-X) Zmacs command 162  
 Compile Region 82  
 Compile Region (m-X) Zmacs command 82  
 Compiler Warnings (m-X) Zmacs command 163  
 Edit Compiler Warnings (m-X) Zmacs command 163  
 Load Compiler Warnings (m-X) Zmacs command 163  
 Dired Complement Dont Delete Flag 140  
 Dired Complement No Reap Flag 139  
 Complete 42  
 Self Insert and Complete 43, 44  
 Complete And Exit 43  
 Complete And Exit If Unique 43  
 COMPLETE completion command 13, 42  
 Completion 12, 42  
 Command completion 42  
 Command name completion 44  
 Impossible-is-OK Completion 43  
 Impossible-is-OK command completion 43  
 Using the HELP key with completion 44  
 Completion Apropos 44  
 c-/ completion command 13, 44  
 c-? completion command 13, 44  
 COMPLETE completion command 13, 42  
 END completion command 13, 43  
 HELP completion command 13  
 RETURN completion command 13  
 SPACE completion command 13  
 Completion Commands 42  
 List Completions 44  
 Standard comtab 191  
 Comtabs 5, 191  
 List contents of a directory 119  
 Exclamation point (!) line continuation indicator 20, 46  
 Additional Notation Conventions 7  
 Entering Converse 30  
 Copy File 121  
 Dired Copy File 137  
 Copy File (m-X) Zmacs command 121  
 Copying files 137  
 Correcting Typos 20  
 Dired Set Generation Retention Count 139  
 Count Lines Page 38  
 Count Lines Region 38  
 Create Directory 124  
 Create Directory (m-X) Zmacs command 123  
 Create Link 122  
 Create Link (m-X) Zmacs command 122  
 Creating a Buffer 27  
 Creating a File 29  
 Creating a Region 75  
 Creating and Saving Buffers and Files 27  
 Creating New Commands: Keyboard Macros 183  
 Current buffer 109  
 Current buffer information 17  
 Save current location in register 78  
 Current meaning of mouse clicks 49  
 Cursor 15



Moving the Cursor 23, 45, 50  
 Relocate cursor 49  
 Cursor Motion by Line 56  
 Cursor Motion by Lisp Expression 53  
 Cursor Motion by Page 59  
 Cursor Motion by Paragraph 58  
 Cursor Motion by Sentence 52  
 Cursor Motion by Word 52  
 Cursor motion within buffer 60  
 Cursor movement commands 23, 51  
 Move cursor to beginning of line 48  
 Customizing the Editor in Init Files 196  
 Customizing the Zmacs Environment 177

**D****D****D**

c-X  
 HELP  
 One Window  
 Select  
 Move to  
 Beginning Of  
 Edit  
 End Of  
 Mark  
 Positioning Window Around  
 Edit  
 Editing the  
 Add Patch Changed  
 Compile Changed  
 Edit Changed  
 Evaluate Changed  
 List  
 Add Patch Changed  
 Compile Changed  
 Edit Changed  
 Evaluate Changed  
 List Changed  
**zwei:**  
 Dired  
 Dired Complement Dont  
 D Dired command 138  
 D Zmacs command 131  
 D Zmacs command 12, 35  
 Decrypt Buffer 108  
 Decrypt Buffer (m-X) Zmacs command 108  
 Default 194  
 Default column position 57  
 Default major mode 182  
 Default Pathnames in Dired 134  
 Default Previous Buffer 105  
 Default Previous Point 76  
 Definition 55  
 Definition 166  
 Definition 56  
 Definition 80  
 Definition 48  
 Definition Command 166  
 definition of a function 11  
 Definitions (m-X) Zmacs command 173  
 Definitions (m-X) Zmacs command 162  
 Definitions (m-X) Zmacs command 167  
 Definitions (m-X) Zmacs command 161  
 Definitions (m-X) Zmacs command 168  
 Definitions of Buffer (m-X) Zmacs command 173  
 Definitions of Buffer (m-X) Zmacs command 162  
 Definitions of Buffer (m-X) Zmacs command 168  
 Definitions of Buffer (m-X) Zmacs command 161  
 Definitions of Buffer (m-X) Zmacs command 169  
**defmajor** 182  
 Deinstall Macro (m-X) Zmacs command 186  
 Delete 138  
 Delete Blank Lines 26, 153  
 Delete commands 25  
 Delete File 122  
 Delete File (m-X) Zmacs command 122  
 Delete Flag 140  
 Delete Forward 25, 66  
 Delete Horizontal Space 26  
 Delete Indentation 26, 152  
 Deleting and Transposing Characters 66  
 Deleting and Transposing Lines 71  
 Deleting and Transposing Lisp Expressions 69

Deleting and Transposing Text 61  
 Deleting and Transposing Words 68  
 Deleting Multiple Versions 138  
 Deleting Sentences 72  
 Deleting vs. Killing 62  
 Accidental deletion 33  
 Large Deletions 33  
 Describe Attribute List 136  
 Dired Describe Attribute List 136  
 Describe Variable (m-X) Zmacs command 193  
 Describe Variable At Point 38  
 Command description 12  
 Indent Differently 150  
 Editing directories 131  
 Create Directory 124  
 Display Directory 119  
 List contents of a directory 119  
 Clean Directory (m-X) Zmacs command 123  
 Create Directory (m-X) Zmacs command 123  
 Dired 131  
 Default Pathnames in Dired 134  
 Dired (m-X) Zmacs command 131  
 Dired Abort 135  
 Dired Apply Function 141  
 Dired Automatic 138, 139  
 Dired Change File Properties 136  
 ! Dired command 140  
 \$ Dired command 139  
 . Dired command 136  
 = Dired command 137  
 ? Dired command 135  
 A Dired command 141  
 ABORT Dired command 135  
 C Dired command 137  
 c-D Dired command 138  
 c-K Dired command 138  
 c-N Dired command 136  
 c-P Dired command 136  
 D Dired command 138  
 E Dired command 137  
 END Dired command 134  
 H Dired command 138  
 HELP Dired command 135  
 K Dired command 138  
 N Dired command 138  
 P Dired command 140  
 Q Dired command 134  
 R Dired command 137  
 RUBOUT Dired command 138  
 SPACE Dired command 136  
 U Dired command 138  
 V Dired command 137  
 Dired Commands 132  
 Dired Complement Dont Delete Flag 140  
 Dired Complement No Reap Flag 139  
 Dired Copy File 137  
 Dired Delete 138  
 Dired Describe Attribute List 136

Dired Edit File 137  
Dired Exit 134  
Dired Hardcopy File 140  
Dired Help 135  
Dired menu 135, 136  
Dired Mode 131  
Dired move point 136  
Dired Next Hog 138  
Dired Next Undumped 140  
Dired Rename File 137  
Dired Reverse Undelete 138  
Dired Set Generation Retention Count 139  
Dired Srccom 137  
Dired Undelete 138  
Dired View File 137  
Display 47  
Display argument list 37  
Display Directory 119  
Display function documentation 37  
Displaying previous keystrokes 36  
Documentation 37  
Brief documentation 37  
Display function documentation 37  
Function documentation 37  
Long documentation 37  
Online documentation for commands 34  
Online documentation for prefixes 35  
Mouse Documentation Line 49  
Tags Tables and Search Domains 94  
Dired Complement Dont Delete Flag 140  
Down Comment Line 158  
Down List 54  
Down Real Line 23, 56, 136

**E**

SELECT  
SELECT  
c-X  
Move To Screen  
Locating Source Code to

**E**

E 30  
E 10  
E Dired command 137  
E Zmacs command 184  
Echo Area 17  
Echoing 17  
Echoing arguments 22  
(ed) function 10, 30  
Edge 48  
Edit 166  
Edit Buffers (m-X) Zmacs command 107  
Edit Callers (m-X) Zmacs command 169  
Multiple Edit Callers (m-X) Zmacs command 170  
Edit Changed Definitions (m-X) Zmacs  
command 167  
Edit Changed Definitions of Buffer (m-X) Zmacs  
command 168  
Edit Compiler Warnings (m-X) Zmacs command 163  
Edit Definition 166  
Edit Definition Command 166  
Dired Edit File 137  
Edit File Warnings (m-X) Zmacs command 163  
zwei: edit-functions function 11

**E**

- Editing a File 29
- Editing directories 131
- Editing Lisp Programs 155
- Major
  - Editing Modes 144
  - Editing the definition of a function 11
  - Editing the source code of a function 11
- Entering File System
  - Editor 30
  - Customizing the
    - Editor in Init Files 196
    - Editor Menu 39
    - Editor menu commands 39
    - Editor Window 15
    - Electric Font Lock Mode (m-X) Zmacs command 180
    - Electric PL1 mode 145
    - Electric Shift Lock mode 18
    - Electric Shift Lock Mode (m-X) Zmacs command 180
    - EMACS Mode (m-X) Zmacs command 181
    - Encrypt Buffer 108
    - Encrypt Buffer (m-X) Zmacs command 108
- Goto
  - End 24, 60
  - Mark
    - End 81
    - END completion command 13, 43
    - END Dired command 134
    - End Kbd Macro 184
    - End Kbd Macro Zmacs command 183
- Moving to
  - end of buffer 60
  - End Of Definition 56
  - End of Line 23, 57
- Using the mouse to
  - enter Zmacs 10
  - Entering Converse 30
  - Entering File System Editor 30
  - Entering Flavor Examiner 30
  - Entering Inspector 30
  - Entering Lisp 30
  - Entering Peek 30
  - Entering Terminal 30
  - Entering Zmacs 10, 30
  - Entering Zmail 30
- Customizing the Zmacs
  - Environment 177
  - Erase backward to start of line 71
  - Erasing Text 25, 66
  - Error recovery 32
  - Escaping from prompts 32
  - Evaluate And Exit 161
  - Evaluate and Replace Into Buffer (m-X) Zmacs command 92, 161
  - Evaluate Buffer (m-X) Zmacs command 160
  - Evaluate Changed Definitions (m-X) Zmacs command 161
  - Evaluate Changed Definitions of Buffer (m-X) Zmacs command 161
  - Evaluate Into Buffer (m-X) Zmacs command 160
  - Evaluate Minibuffer 160
  - Evaluate Region 160
  - Evaluate Region (m-X) Zmacs command 160, 161
  - Evaluate Region Verbose 160, 161
  - Evaluation and Compilation 160

Entering Flavor	Evaluation commands	160
Selecting, Listing, and	Examiner	30
	Examining Buffers	104
	Exchange Characters	66
	Exchange Lines	71
	Exchange Regions	82
	Exchange Sexps	70
	Exchange Words	68
	Exclamation point (!) line continuation indicator	20, 46
Compile and	Exit	162
Complete And	Exit	43
Dired	Exit	134
Evaluate And	Exit	161
Complete And	Exit If Unique	43
	Exiting Zmacs	30
Macro	Expand Expression	165
Macro	Expand Expression (m-X) Zmacs command	165
Macro	Expand Expression All	165
	Expanding Expressions	165
Cursor Motion by Lisp	Expression	53
Macro Expand	Expression	165
Mark Lisp	expression	80
Top-level	expression	55
Macro Expand	Expression (m-X) Zmacs command	165
Macro Expand	Expression All	165
Deleting and Transposing Lisp	Expressions	69
Expanding	Expressions	165
Parenthesizing	Expressions	164
Point Motion Among Top-Level	Expressions	55
	Extended Command	5
Any	Extended Command	6
	Extended command key bindings	36
	Extended commands	4, 5, 35

## F

## F

## F

SELECT	F	30
c-X	F Zmacs command	27, 180
	Fast Where Am I	37
	File	19
Copy	File	121
Creating a	File	29
Delete	File	122
Dired Copy	File	137
Dired Edit	File	137
Dired Hardcopy	File	140
Dired Rename	File	137
Dired View	File	137
Editing a	File	29
Find	File	27, 108
Naming a	File	109
Rename	File	120
Save	File	27, 109
Visit	File	109
Write	File	27, 109
Append To	File (m-X) Zmacs command	112
Compile	File (m-X) Zmacs command	162

Copy	File (m-X) Zmacs command	121
Delete	File (m-X) Zmacs command	122
Hardcopy	File (m-X) Zmacs command	120
Insert	File (m-X) Zmacs command	112
Prepend To	File (m-X) Zmacs command	112
Reap	File (m-X) Zmacs command	122
Rename	File (m-X) Zmacs command	120
Set Patch	File (m-X) Zmacs command	128
View	File (m-X) Zmacs command	120
Set commands for	file and buffer attributes	127
Backspace	file attribute	128
Base	file attribute	128
Lowercase	file attribute	128
Nofill	file attribute	129
Patch-File	file attribute	129
Tab-Width	file attribute	130
Vsp	file attribute	130
	File Attribute Checking	126
Warnings about	file attribute lists	126
	File attributes	125
Buffer and	File Attributes	125
	File backup flag	140
	File flags	103
Find	File In Fundamental Mode (m-X) Zmacs command	110
	File Manipulation Commands	119
Set Visited	File Name (m-X) Zmacs command	110
Dired Change	File Properties	136
View	File Properties	120
Change	File Properties (m-X) Zmacs command	123
View	File Properties (m-X) Zmacs command	120
Entering	File System Editor	30
	File types and major modes	182
	File types of major modes	182
	File versions	102
Comparing	file versions	137
Edit	File Warnings (m-X) Zmacs command	163
Find	File Zmacs command	28, 29
Save	File Zmacs command	28
Write	File Zmacs command	29
<b>*fs:</b>	<b>file-type-mode-alist*</b> Lisp variable	182
Applying Functions to	Files	27
Association of buffers with	Files	141
Commands for manipulating	files	27
Copying	files	119
Creating and Saving Buffers and	files	137
Customizing the Editor in Init	Files	27
Init	Files	196
Manipulating Buffers and	Files	196
Protecting	Files	101
Renaming	Files	139
Using the mouse with List	Files	137
Working With Buffers and	Files	119
List	Files	102
Save All	Files (m-X) Zmacs command	119
Comparing	Files (m-X) Zmacs command	108
Find	Files and Buffers	113
	Files in Tag Table (m-X) Zmacs command	97

Association of	files with buffers	27
Set	Fill Column	180
Auto	Fill Lisp Comments Mode (c-m-X) Zmacs command	180
Auto	Fill mode	18, 129, 179
Auto	Fill Mode (m-X) Zmacs command	180
	Fill Paragraph	83
Set	Fill Prefix	83
	Fill Region	83
	Find File	27, 108
	Find File In Fundamental Mode (m-X) Zmacs command	110
	Find File Zmacs command	28, 29
	Find Files in Tag Table (m-X) Zmacs command	97
	Find Unbalanced Parentheses (m-X) Zmacs command	38
Zmacs commands for	finding out about flavors	202
Zmacs commands for	finding out about Lisp	201
Zmacs commands for	finding out about the state of buffers	201
Zmacs commands for	finding out about the state of Zmacs	201
	Finding Out About Zmacs Commands	34
	Finding source code	156
	Finding the right command	35
	Finish Patch (m-X) Zmacs command	174
Dired Complement Dont Delete	Flag	140
Dired Complement No Reap	Flag	139
File backup	flag	140
Modification	flag	103
+	flag in Zmacs	103
File	flags	103
Entering	Flavor Examiner	30
Zmacs commands for finding out about	flavors	202
Electric	Font Lock Mode (m-X) Zmacs command	180
Set	Fonts (m-X) Zmacs command	128
Response	Format	41
	Forward	23, 51
Delete	Forward	25, 66
	Forward character	51
	Forward List	53
	Forward Paragraph	23, 58
	Forward Sentence	23, 53
	Forward Sexp	54
	Forward Up List	55
	Forward Word	23, 52
Multiple Query Replace	From Buffer (m-X) Zmacs command	92
	<b>*fs:file-type-mode-alist*</b> Lisp variable	182
Dired Apply	Function	141
(ed)	function	10, 30
Editing the definition of a	function	11
Editing the source code of a	function	11
<b>zwei:edit-functions</b>	function	11
	Function Documentation	37
Display	function documentation	37
	<b>*Function-Specs-to-Edit-r*</b> buffer	11
Zmacs support	functions	98
Applying	Functions to Files	141
	Fundamental mode	18, 144
Find File In	Fundamental Mode (m-X) Zmacs command	110

## G

**G**

Dired Set  
Open

**ibase**  
**zwei:\*set-attribute-update-list\***  
Set

Generation Retention Count 139  
Get Register 78  
Getting Help 31  
Getting Help in Zmacs 12  
Getting Out of Trouble 32  
Getting Started 9  
Global search and replace 94  
global variable 128  
global variable 128  
Goal Column 57  
Goto Beginning 24, 60  
Goto End 24, 60  
Grow Window 117

## G

## H

**H**

c-X  
Dired  
Dired  
Getting  
Response

Getting  
Using the

Buffer  
Command  
Kill  
Yanking in the command  
Yanking in the kill

Dired Next  
Delete

H Dired command 138  
H Zmacs command 81  
Hardcopy Buffer (m-X) Zmacs command 108  
Hardcopy File 140  
Hardcopy File (m-X) Zmacs command 120  
HELP 13, 34  
Help 135  
Help 31  
Help 41  
HELP ? Zmacs command 12  
HELP A Zmacs command 12, 35  
HELP C Zmacs command 12, 34  
HELP completion command 13  
HELP D Zmacs command 12, 35  
HELP Dired command 135  
Help in Zmacs 12  
HELP key 12, 34  
HELP key with completion 44  
HELP L Zmacs command 12, 36  
HELP SPACE Zmacs command 12  
HELP U Zmacs command 12, 36  
HELP V Zmacs command 12, 36, 193  
HELP W Zmacs command 12  
History 104  
history 33  
history 33, 65  
history 14  
history 13  
History list 13  
Hog 138  
Horizontal Space 26  
How to Specify Zmacs Variable Settings 193

## H



Fast Where Am	I 37
SELECT	I 30
Where Am	I 37
Complete And Exit	<b>ibase</b> global variable 128
	If Unique 43
	Impossible-is-OK command completion 43
	Impossible-is-OK Completion 43
	Incremental Search 86
Reverse	Incremental Search 87
	Indent Differently 150
	Indent For Comment 157
	Indent for Lisp 150
	Indent New Comment Line 158
	Indent New Line 151
	Indent Region 151
	Indent Rigidly 152
	Indent Sexp 151
	Indentation 150
Back To	Indentation 151
Changing Case and	Indentation 147
Delete	Indentation 26, 152
This	Indentation 152
* buffer status	indicator 19
Exclamation point (!) line continuation	indicator 20, 46
Semicolon (;) comment	indicator 157
Current buffer	information 17
Customizing the Editor in	Init Files 196
Uppercase	Init Files 196
Self	Initial 148
	Insert and Complete 43, 44
	Insert Buffer (m-X) Zmacs command 112
	Insert File (m-X) Zmacs command 112
	Insert Matching parentheses 164
	Insert Tab 150
	Insert text from register into buffer 78
	Inserting Characters 20
	Inserting Text 20
Appending, Prepending, and	Inserting Text 112
Entering	Inspector 30
	Install Command (m-X) Zmacs command 192
	Install Macro (m-X) Zmacs command 186
Zmacs commands for	interacting with Lisp 202
	Introduction 1, 156
	Introduction to Zmacs 4
	Invoking Zmacs 10
What	is a Region? 74

J

J

Jump to Saved Position 78

J

## K

## K

## K

c-X K Dired command 138  
 Call Last K Zmacs command 110  
 End Kbd Macro 184  
 Start Kbd Macro 183, 184  
 Name Last Kbd Macro (m-X) Zmacs command 185  
 View Kbd Macro (m-X) Zmacs command 184  
 Kbd Macro Query 185  
 End Kbd Macro Zmacs command 183  
 HELP key 12, 34  
 RUBOUT key 20  
 SELECT key 30  
 Set Key (m-X) Zmacs command 191  
 Key Bindings 191  
 Assign key bindings 192  
 Command key bindings 12  
 Extended command key bindings 36  
 Using the RETURN key in the minibuffer 41  
 Using the HELP key with completion 44  
 Creating New Commands: Keyboard Macros 183  
 Sort Via Keyboard Macros 186  
 Sort Via Keyboard Macros (m-X) Zmacs command 100  
 Shift keys 5  
 Keystrokes 4  
 Displaying previous keystrokes 36  
 List the last sixty keystrokes 36  
 Append Next Kill 65  
 Query Replace Last Kill (m-X) Zmacs command 92  
 Kill Backward Up List (c-m-X) Zmacs command 69  
 Kill Buffer 110  
 Kill commands 25  
 Kill Comment 157  
 Kill history 33, 65  
 Yanking in the kill history 13  
 Kill Line 25, 71  
 Kill Or Save Buffers (m-X) Zmacs command 110  
 Kill Region 25, 82  
 Kill Sentence 25, 72  
 Backward Kill Sentence 25, 72  
 Kill Sexp 25, 69  
 Backward Kill Sexp 25, 69  
 Kill Some Buffers (m-X) Zmacs command 110  
 Kill Word 25, 68  
 Backward Kill Word 25, 68  
 Deleting vs. Killing 62

## L

## L

## L

SELECT L 30  
 c-X L Zmacs command 38  
 HELP L Zmacs command 12, 36  
 L2:Move to point mouse click 49  
 L:Move point mouse click 49  
 Large Deletions 33  
 Cancel last command 32  
 Call Last Kbd Macro 184

Name	Last Kbd Macro (m-X) Zmacs command	185
Query Replace	Last Kill (m-X) Zmacs command	92
Repeat	Last Minibuffer Command	41, 64
List the	last sixty commands	36
List the	last sixty keystrokes	36
	Leaving Zmacs	30
Query Replace	Let Binding (m-X) Zmacs command	92
Abort At Top	Level	32
Point motion along nesting	level	54
Point motion between nesting	levels	54
Beginning of	Line	23, 57
Breaking a	line	20
Center	Line	150
Cursor Motion by	Line	56
Down Comment	Line	158
Down Real	Line	23, 56, 136
End of	Line	23, 57
Erase backward to start of	line	71
Indent New	Line	151
Indent New Comment	Line	158
Kill	Line	25, 71
Mode	line	17
Mouse Documentation	Line	49
Move cursor to beginning of	line	48
Move To Specified	Line	48
Split	Line	153
Starting a New	Line	20
Up Comment	Line	158
Up Real	Line	23, 56, 136
Exclamation point (!)	line continuation indicator	20, 46
	LINE Zmacs command	151
Delete Blank	Lines	26, 153
Deleting and Transposing	Lines	71
Exchange	Lines	71
Merging	lines	20
Wraparound	Lines	46
Wrapping	Lines	20
Sort	Lines (m-X) Zmacs command	100
Count	Lines Page	38
Count	Lines Region	38
Create	Link	122
Create	Link (m-X) Zmacs command	122
Entering	Lisp	30
Indent for	Lisp	150
Zmacs commands for finding out about	Lisp	201
Zmacs commands for interacting with	Lisp	202
Auto Fill	Lisp Comments Mode (c-m-X) Zmacs command	180
Cursor Motion by	Lisp Expression	53
Mark	Lisp expression	80
Deleting and Transposing	Lisp Expressions	69
	Lisp mode	18, 144, 156
	Lisp Mode (m-X) Zmacs command	156
Editing	Lisp Programs	155
<b>*fs:file-type-mode-alist*</b>	Lisp variable	182
<b>*zwei:~major-mode-translations*</b>	Lisp variable	182
Attribute	list	125
Backward	List	53, 54

- Backward Up List 55
- Describe Attribute List 136
- Dired Describe Attribute List 136
- Display argument list 37
- Down List 54
- Forward List 53
- Forward Up List 55
- History list 13
- Kill Backward Up List (c-m-X) Zmacs command 69
- Reparse Attribute List (m-X) Zmacs command 127
- Reparse Attribute List (m-X) Zmacs command 125
- Update Attribute List (m-X) Zmacs command 125
- List Buffers 106
- List Buffers Zmacs command 103
- List Callers (m-X) Zmacs command 170
- List Callers (m-X) Zmacs command 171
- List Changed Definitions of Buffer (m-X) Zmacs command 169
- List Completions 44
- List contents of a directory 119
- List Definitions (m-X) Zmacs command 168
- List Files 119
- List Files (m-X) Zmacs command 119
- List Tag Tables (m-X) Zmacs command 95
- List the last sixty commands 36
- List the last sixty keystrokes 36
- List Variables (m-X) Zmacs command 193
- Listing, and Examining Buffers 104
- lists 126
- Warnings about file attribute lists 126
- Load Compiler Warnings (m-X) Zmacs command 163
- Local names of commands 6
- Locating and Replacing Strings Automatically 90
- Locating Source Code to Edit 166
- location in register 78
- Save current Lock mode 18
- Electric Shift Lock Mode (m-X) Zmacs command 180
- Electric Font Lock Mode (m-X) Zmacs command 180
- Electric Shift Long Documentation 37
- Set Lowercase (m-X) Zmacs command 128
- Lowercase Code in Region 148
- Lowercase file attribute 128
- Lowercase Region 148
- Lowercase Word 148

**M****M****M**

- SELECT M 30
- m-X Zmacs command 90
  - m-( Zmacs command 164
  - m-) Zmacs command 56
  - m-. Zmacs command 166
  - m-; Zmacs command 157
  - m-< Zmacs command 24, 60
  - m-= Zmacs command 38
  - m-> Zmacs command 24, 60
  - m-@ Zmacs command 80

	m-A Zmacs command	23, 53
	m-B Zmacs command	23, 52
	m-C Zmacs command	148
	m-D Zmacs command	25, 68
	m-E Zmacs command	53
	m-ESCAPE Zmacs command	160
	m-F Zmacs command	23, 52
	m-H Zmacs command	80
	m-K Zmacs command	25, 72
	m-L Zmacs command	148
	m-LINE Zmacs command	158
	m-M Zmacs command	151
	m-N Zmacs command	158
	m-O Zmacs command	152
	m-P Zmacs command	158
	m-R Zmacs command	48
	m-RETURN Zmacs command	151
	m-RUBOUT Zmacs command	25, 68
	m-S Zmacs command	150
	m-sh-C Zmacs command	162
	m-sh-D Zmacs command	37
	m-sh-E Zmacs command	161
	m-T Zmacs command	68
	m-U Zmacs command	148
	m-V Zmacs command	24, 48
	m-W Zmacs command	82
Abort Patch	(m-X) Zmacs command	174
Add Patch	(m-X) Zmacs command	173
Add Patch Changed Definitions	(m-X) Zmacs command	173
Add Patch Changed Definitions of Buffer	(m-X) Zmacs command	173
Append To File	(m-X) Zmacs command	112
Arglist	(m-X) Zmacs command	37
Atom Query Replace	(m-X) Zmacs command	93
Atom Word Mode	(m-X) Zmacs command	180
Auto Fill Mode	(m-X) Zmacs command	180
Change File Properties	(m-X) Zmacs command	123
Clean Directory	(m-X) Zmacs command	123
Compile Buffer	(m-X) Zmacs command	162
Compile Changed Definitions	(m-X) Zmacs command	162
Compile Changed Definitions of Buffer	(m-X) Zmacs command	162
Compile File	(m-X) Zmacs command	162
Compile Region	(m-X) Zmacs command	82
Compiler Warnings	(m-X) Zmacs command	163
Copy File	(m-X) Zmacs command	121
Create Directory	(m-X) Zmacs command	123
Create Link	(m-X) Zmacs command	122
Decrypt Buffer	(m-X) Zmacs command	108
Deinstall Macro	(m-X) Zmacs command	186
Delete File	(m-X) Zmacs command	122
Describe Variable	(m-X) Zmacs command	193
Dired	(m-X) Zmacs command	131
Edit Buffers	(m-X) Zmacs command	107
Edit Callers	(m-X) Zmacs command	169
Edit Changed Definitions	(m-X) Zmacs command	167
Edit Changed Definitions of Buffer	(m-X) Zmacs command	168
Edit Compiler Warnings	(m-X) Zmacs command	163
Edit File Warnings	(m-X) Zmacs command	163

Electric Font Lock Mode	(m-X) Zmacs command	180
Electric Shift Lock Mode	(m-X) Zmacs command	180
EMACS Mode	(m-X) Zmacs command	181
Encrypt Buffer	(m-X) Zmacs command	108
Evaluate and Replace Into Buffer	(m-X) Zmacs command	92, 161
Evaluate Buffer	(m-X) Zmacs command	160
Evaluate Changed Definitions	(m-X) Zmacs command	161
Evaluate Changed Definitions of Buffer	(m-X) Zmacs command	161
Evaluate Into Buffer	(m-X) Zmacs command	160
Evaluate Region	(m-X) Zmacs command	160, 161
Find File In Fundamental Mode	(m-X) Zmacs command	110
Find Files in Tag Table	(m-X) Zmacs command	97
Find Unbalanced Parentheses	(m-X) Zmacs command	38
Finish Patch	(m-X) Zmacs command	174
Hardcopy Buffer	(m-X) Zmacs command	108
Hardcopy File	(m-X) Zmacs command	120
Insert Buffer	(m-X) Zmacs command	112
Insert File	(m-X) Zmacs command	112
Install Command	(m-X) Zmacs command	192
Install Macro	(m-X) Zmacs command	186
Kill Or Save Buffers	(m-X) Zmacs command	110
Kill Some Buffers	(m-X) Zmacs command	110
Lisp Mode	(m-X) Zmacs command	156
List Callers	(m-X) Zmacs command	170
List Changed Definitions of Buffer	(m-X) Zmacs command	169
List Definitions	(m-X) Zmacs command	168
List Files	(m-X) Zmacs command	119
List Tag Tables	(m-X) Zmacs command	95
List Variables	(m-X) Zmacs command	193
Load Compiler Warnings	(m-X) Zmacs command	163
Macro Expand Expression	(m-X) Zmacs command	165
Multiple Edit Callers	(m-X) Zmacs command	170
Multiple List Callers	(m-X) Zmacs command	171
Multiple Query Replace	(m-X) Zmacs command	91
Multiple Query Replace From Buffer	(m-X) Zmacs command	92
Name Last Kbd Macro	(m-X) Zmacs command	185
Overwrite Mode	(m-X) Zmacs command	181
Prepend To File	(m-X) Zmacs command	112
Query Replace	(m-X) Zmacs command	90
Query Replace Last Kill	(m-X) Zmacs command	92
Query Replace Let Binding	(m-X) Zmacs command	92
Reap File	(m-X) Zmacs command	122
Rename Buffer	(m-X) Zmacs command	108
Rename File	(m-X) Zmacs command	120
Reparsing Attribute List	(m-X) Zmacs command	125
Replace String	(m-X) Zmacs command	90
Revert Buffer	(m-X) Zmacs command	109
Save All Files	(m-X) Zmacs command	108
Select All Buffers As Tag Table	(m-X) Zmacs command	94
Select System As Tag Table	(m-X) Zmacs command	95
Select Tag Table	(m-X) Zmacs command	94
Set Backspace	(m-X) Zmacs command	128
Set Base	(m-X) Zmacs command	128
Set Fonts	(m-X) Zmacs command	128
Set Key	(m-X) Zmacs command	191
Set Lowercase	(m-X) Zmacs command	128
Set Nofill	(m-X) Zmacs command	128
Set Package	(m-X) Zmacs command	126, 127, 128

Set Patch File	(m-X) Zmacs command	128
Set Tab Width	(m-X) Zmacs command	128
Set Variable	(m-X) Zmacs command	194
Set Visited File Name	(m-X) Zmacs command	110
Set Vsp	(m-X) Zmacs command	128
Sort Lines	(m-X) Zmacs command	100
Sort Paragraphs	(m-X) Zmacs command	100
Sort Via Keyboard Macros	(m-X) Zmacs command	100
Source Compare	(m-X) Zmacs command	113
Source Compare Merge	(m-X) Zmacs command	113
Start Patch	(m-X) Zmacs command	172
Start Private Patch	(m-X) Zmacs command	172
Tags Multiple Query Replace	(m-X) Zmacs command	96
Tags Query Replace	(m-X) Zmacs command	95
Tags Search	(m-X) Zmacs command	95
Trace	(m-X) Zmacs command	38
Uncomment Region	(m-X) Zmacs command	159
Update Attribute List	(m-X) Zmacs command	125
Uppercase Code in Region	(m-X) Zmacs command	148
View Buffer	(m-X) Zmacs command	107
View File	(m-X) Zmacs command	120
View File Properties	(m-X) Zmacs command	120
View Kbd Macro	(m-X) Zmacs command	184
Visit Tag Table	(m-X) Zmacs command	97
Word Abbrev Mode	(m-X) Zmacs command	181
	m-Y yank command	13, 14, 64
	m-Z Zmacs command	162
	m-[ Zmacs command	23, 58
	m-\ Zmacs command	26
	m-] Zmacs command	23, 58
	m-^ Zmacs command	26, 152
	M2:Save/Kill/Yank mouse click	49
	M:Mark thing mouse click	49
Call Last Kbd	Macro	184
End Kbd	Macro	184
Start Kbd	Macro	183, 184
Deinstall	Macro (m-X) Zmacs command	186
Install	Macro (m-X) Zmacs command	186
Name Last Kbd	Macro (m-X) Zmacs command	185
View Kbd	Macro (m-X) Zmacs command	184
	Macro Expand Expression	165
	Macro Expand Expression (m-X) Zmacs command	165
	Macro Expand Expression All	165
	Macro Query	185
Kbd	Macro Zmacs command	183
End Kbd	Macro Zmacs command	183
Creating New Commands: Keyboard	Macros	183
Sort Via Keyboard	Macros	186
Sort Via Keyboard	Macros (m-X) Zmacs command	100
	Macsyma mode	18, 144
	Major Editing Modes	144
Default	major mode	182
Setting the	Major Mode	143
	Major Modes	182
File types and	major modes	182
File types of	major modes	182
User-defined	major modes	182
	Major-mode	18

<b>zwei:</b>	<b>*major-mode-translations*</b> Lisp variable	182
	Make region	49
	Make Room	153
Commands for	Manipulating Buffers and Files	101
File	manipulating files	119
Organization of this	Manipulation Commands	119
Region Right	manual	2
	Margin Mode	194
	Mark	74
Set Pop	Mark	76
Swap Point And	Mark	77
	Mark Beginning	81
	Mark buffer	81
	Mark Definition	80
	Mark End	81
	Mark Lisp expression	80
	Mark Page	81
	Mark Paragraph	80
	Mark region	49
Commands to	Mark Regions	80
	Mark Sexp	80
M:	Mark thing mouse click	49
	Mark Whole	81
	Mark Word	80
Region	Marking Mode	194
	Marking text	80
Insert	Matching parentheses	164
Current	meaning of mouse clicks	49
Dired	menu	135, 136
System	menu	30
The Editor	Menu	39
Using the mouse to call up System	menu	30
Editor	menu commands	39
R2:System	menu mouse click	50
R:	Menu mouse click	50
Source Compare	Merge	114
Source Compare	Merge (m-X) Zmacs command	113
	Merging lines	20
	Midas mode	145
	Minibuffer	12, 17
Evaluate	Minibuffer	160
More on the	Minibuffer	41
Using the RETURN key in the	minibuffer	41
Repeat Last	Minibuffer Command	41, 64
	Minibuffer Prompts	32
	Minor Modes	179
	Minor-mode	18
	Mode	17
Auto Fill	mode	18, 129, 179
Bollo	mode	145
Default major	mode	182
Dired	Mode	131
Electric PL1	mode	145
Electric Shift Lock	mode	18
Fundamental	mode	18, 144
Lisp	mode	18, 144, 156
Macsyma	mode	18, 144
Midas	mode	145



PL1	mode	145
Region Marking	Mode	194
Region Right Margin	Mode	194
Setting the Major	Mode	143
Teco	mode	145
Text	mode	18, 144
Two window	mode	117
Word Abbrev	mode	18
Auto Fill Lisp Comments	Mode (c-m-X) Zmacs command	180
Atom Word	Mode (m-X) Zmacs command	180
Auto Fill	Mode (m-X) Zmacs command	180
Electric Font Lock	Mode (m-X) Zmacs command	180
Electric Shift Lock	Mode (m-X) Zmacs command	180
EMACS	Mode (m-X) Zmacs command	181
Find File In Fundamental	Mode (m-X) Zmacs command	110
Lisp	Mode (m-X) Zmacs command	156
Overwrite	Mode (m-X) Zmacs command	181
Word Abbrev	Mode (m-X) Zmacs command	181
	Mode line	17
File types and major	modes	182
File types of major	modes	182
Major	Modes	182
Major Editing	Modes	144
Minor	Modes	179
User-defined major	modes	182
	Modification flag	103
	Modified Two Windows	117
	More on the Minibuffer	41
	Motion	50
Point	motion along nesting level	54
Point	Motion Among Top-Level Expressions	55
Point	motion between nesting levels	54
	Motion by character	51
Cursor	Motion by Line	56
Cursor	Motion by Lisp Expression	53
	Motion by Page	59
Cursor	Motion by Page	59
	Motion by Paragraph	58
Cursor	Motion by Paragraph	58
Cursor	Motion by Sentence	52
Cursor	Motion by Word	52
	Motion Commands	51
	Motion with Respect to the Whole Buffer	60
Cursor	motion within buffer	60
	Mouse	10, 49, 106, 114, 166
Using the	Mouse	49
	[Mouse (R)]	44
L2:Move to point	mouse click	49
L:Move point	mouse click	49
M2:Save/Kill/Yank	mouse click	49
M:Mark thing	mouse click	49
R2:System menu	mouse click	50
R:Menu	mouse click	50
Current meaning of	mouse clicks	49
	Mouse Documentation Line	49
Using the	mouse to call up System menu	30
Using the	mouse to enter Zmacs	10
Using the	mouse with List Files	119

Mouse-sensitive Zmacs commands 50  
 Move cursor to beginning of line 48  
 Move Over ) 56  
 Dired move point 136  
 L: Move point mouse click 49  
 Move to Default Previous Point 76  
 L2: Move to point mouse click 49  
 Move to Previous Point 76  
 Move To Screen Edge 48  
 Move To Specified Line 48  
 Cursor movement commands 23, 51  
 Moving the Cursor 23, 45, 50  
 Moving to end of buffer 60  
 Multiple buffers 102  
 Multiple Edit Callers (m-X) Zmacs command 170  
 Multiple List Callers (m-X) Zmacs command 171  
 Multiple Query Replace (m-X) Zmacs command 91  
 Tags Multiple Query Replace (m-X) Zmacs command 96  
 Multiple Query Replace From Buffer (m-X) Zmacs  
 command 92  
 Deleting Multiple Versions 138  
 Multiple windows 102

## N

## N

## N

N Dired command 138  
 Set Visited File Name (m-X) Zmacs command 110  
 Command name completion 44  
 Name Last Kbd Macro (m-X) Zmacs command 185  
 Buffer names 18  
 Command Names 4  
 Names of commands 5, 34  
 Local names of commands 6  
 Naming a File 109  
 Negative Numeric Arguments 51  
 Point motion along nesting level 54  
 Point motion between nesting levels 54  
 Creating New Commands: Keyboard Macros 183  
 Indent New Comment Line 158  
 Indent New Line 151  
 Starting a New Line 20  
 Newline characters 20  
 Newlines 20, 56  
 Dired Next Hog 138  
 Append Next Kill 65  
 Next Page 23, 59  
 Next Possibility 98, 168  
 Next Screen 24, 47  
 Dired Next Undumped 140  
 Dired Complement No Reap Flag 139  
 Set Nofill (m-X) Zmacs command 128  
 Nofill file attribute 129  
 Additional Notation Conventions 7  
 Nullifying prefixes 32  
 Quadruple Numeric Arg 22  
 Numeric arguments 20, 22, 34, 51  
 Negative Numeric Arguments 51

O

O

O

c-X O Zmacs command 118  
One Window 118  
One Window Default 194  
Online documentation for commands 34  
Online documentation for prefixes 35  
Open Get Register 78  
Organization of the Screen 15  
Organization of this manual 2  
Other Window 118  
Other Window 118  
Scroll  
Zmacs commands for finding  
Zmacs commands for finding  
Zmacs commands for finding  
Zmacs commands for finding  
Finding  
Getting  
Comment  
out about flavors 202  
out about Lisp 201  
out about the state of buffers 201  
out about the state of Zmacs 201  
Out About Zmacs Commands 34  
Out of Trouble 32  
Out Region 159  
Overview 2, 46, 178  
Overwrite Mode (m-X) Zmacs command 181

P

P

P

SELECT P 30  
P Dired command 140  
Set  
Package (m-X) Zmacs command 126, 127, 128  
Packages 125, 126  
Page 38  
Page 59  
Page 81  
Page 59  
Page 23, 59  
Page 24, 59  
paragraph 83  
Paragraph 23, 58  
Paragraph 58  
Paragraph 83  
Paragraph 23, 58  
Paragraph 80  
Paragraph 58  
Paragraphs (m-X) Zmacs command 100  
parentheses 164  
Parentheses (m-X) Zmacs command 38  
Parentheses When Saving 195  
Parenthesizing Expressions 164  
Patch (m-X) Zmacs command 174  
Patch (m-X) Zmacs command 173  
Patch (m-X) Zmacs command 174  
Patch (m-X) Zmacs command 172  
Patch (m-X) Zmacs command 172  
Patch Changed Definitions (m-X) Zmacs  
command 173  
Patch Changed Definitions of Buffer (m-X) Zmacs  
command 173  
Patch File (m-X) Zmacs command 128  
Patch-File file attribute 129  
Patching 172

Default	Pathnames in Dired	134
Entering	Peek	30
	Pixel	180
	PL1 mode	145
Electric	PL1 mode	145
	Point	15, 37, 54, 74
Describe Variable At	Point	38
Dired move	point	136
Move to Default Previous	Point	76
Move to Previous	Point	76
Exclamation	point (!) line continuation indicator	20, 46
Swap	Point And Mark	77
	Point motion along nesting level	54
	Point Motion Among Top-Level Expressions	55
	Point motion between nesting levels	54
L2:Move to	point mouse click	49
L:Move	point mouse click	49
	Point-pdl	75
Buffer	pointers	74
Yank	Pop	64
Set	Pop Mark	76
Default column	position	57
Jump to Saved	Position	78
Save	Position	78
	Positioning Window Around Definition	48
	Possibilities	98
Next	Possibility	98, 168
	Possibility Buffers	98
Set Fill	Prefix	83
	Prefix character commands	5
	Prefix Commands	35
	Prefixes	32
Nullifying	prefixes	32
Online documentation for	prefixes	35
	Prepend To File (m-X) Zmacs command	112
Appending,	Prepending, and Inserting Text	112
Select	Previous Buffer	105
Select Default	Previous Buffer	105
Displaying	previous keystrokes	36
	Previous Page	24, 59
Move to	Previous Point	76
Move to Default	Previous Point	76
	Previous Screen	24, 48
Return to	previous window	30
Start	Private Patch (m-X) Zmacs command	172
Editing Lisp	Programs	155
Escaping from	prompts	32
Minibuffer	Prompts	32
Dired Change File	Properties	136
View File	Properties	120
Change File	Properties (m-X) Zmacs command	123
View File	Properties (m-X) Zmacs command	120
	Protecting Files	139
	Put Register	78

## Q

Q  
 c-X Q Dired command 134  
 Q Zmacs command 185  
 Kbd Macro Quadruple Numeric Arg 22  
 Query 185  
 Query Replace 90  
 Query Replace (m-X) Zmacs command 90  
 Atom Query Replace (m-X) Zmacs command 93  
 Multiple Query Replace (m-X) Zmacs command 91  
 Tags Query Replace (m-X) Zmacs command 95  
 Tags Multiple Query Replace (m-X) Zmacs command 96  
 Multiple Query Replace From Buffer (m-X) Zmacs  
 command 92  
 Query Replace Last Kill (m-X) Zmacs command 92  
 Query Replace Let Binding (m-X) Zmacs  
 command 92  
 Quick Arglist 37

## R

R  
 [Mouse R Dired command 137  
 (R)] 44  
 R2:System menu mouse click 50  
 R:Menu mouse click 50  
 Down Real Line 23, 56, 136  
 Up Real Line 23, 56, 136  
 Dired Complement No Reap File (m-X) Zmacs command 122  
 Reap Flag 139  
 Reaping 139  
 Recenter Window 47  
 Error recovery 32  
 Redisplay 47  
 Redisplaying the Window 47  
 Region 74  
 Adjust region 83  
 Changing Case of Region 148  
 Comment Out Region 159  
 Compile Region 82  
 Count Lines Region 38  
 Creating a Region 75  
 Evaluate Region 160  
 Fill Region 83  
 Indent Region 151  
 Kill Region 25, 82  
 Lowercase Region 148  
 Lowercase Code in Region 148  
 Make region 49  
 Mark region 49  
 Save Region 82  
 Two Windows Showing Region 117  
 Uppercase Region 148  
 Compile Region (m-X) Zmacs command 82  
 Evaluate Region (m-X) Zmacs command 160, 161  
 Uncomment Region (m-X) Zmacs command 159  
 Uppercase Code in Region (m-X) Zmacs command 148  
 Region Marking Mode 194  
 Region Right Margin Mode 194

- Evaluate Region Verbose 160, 161
- Region-Manipulating Commands 82
- What is a Region? 74
- Commands to Mark Regions 80
- Commenting Regions 159
- Exchange Regions 82
- Working with Regions 73
- Open Get Register 78
- Put Register 78
- Save current location in register 78
- Insert text from register into buffer 78
- Registers 78
- Reinitializing Zmacs 11
- Relocate cursor 49
- Rename Buffer (m-X) Zmacs command 108
- Rename File 120
- Dired Rename File 137
- Rename File (m-X) Zmacs command 120
- Renaming Files 137
- Reparse Attribute List (m-X) Zmacs command 127
- Reparse Attribute List (m-X) Zmacs command 125
- Repeat Last Minibuffer Command 41, 64
- Global search and replace 94
- Query Replace 90
- Atom Query Replace (m-X) Zmacs command 93
- Multiple Query Replace (m-X) Zmacs command 91
- Query Replace (m-X) Zmacs command 90
- Tags Multiple Query Replace (m-X) Zmacs command 96
- Tags Query Replace (m-X) Zmacs command 95
- Multiple Query Replace From Buffer (m-X) Zmacs command 92
- Evaluate and Query Replace Into Buffer (m-X) Zmacs command 92, 161
- Query Replace Last Kill (m-X) Zmacs command 92
- Query Replace Let Binding (m-X) Zmacs command 92
- Replace String 90
- Replace String (m-X) Zmacs command 90
- Replacing Strings Automatically 90
- Locating and Searching, Replacing, and Sorting 85
- Reposition Window 48
- Motion with Respect to the Whole Buffer 60
- Cancel response 32
- Response Format 41
- Response Help 41
- Restoring text 33
- Dired Set Generation Retention Count 139
- Carriage return 5
- Using the RETURN completion command 13
- RETURN key in the minibuffer 41
- Return to previous window 30
- RETURN Zmacs command 43
- Reverse Incremental Search 87
- Dired Reverse Undelete 138
- Finding the Revert Buffer (m-X) Zmacs command 109
- Region right command 35
- Indent Right Margin Mode 194
- Make Rigidly 152
- Room 153
- Rubout 25, 66
- RUBOUT Dired command 138

RUBOUT key 20  
 RUBOUT Zmacs character 66  
 RUBOUT Zmacs command 25, 33  
 c-X RUBOUT Zmacs command 25, 72

**S****S****S**

c-X S Zmacs command 27  
 Kill Or Save All Files (m-X) Zmacs command 108  
 Save Buffers (m-X) Zmacs command 110  
 Save current location in register 78  
 Save File 27, 109  
 Save File Zmacs command 28  
 Save Position 78  
 Save Region 82  
 M2: Save/Kill/Yank mouse click 49  
 Jump to Saved Position 78  
 Check Unbalanced Parentheses When Saving 195  
 Creating and Saving Buffers and Files 27  
 Next Screen 24, 47  
 Organization of the Screen 15  
 Previous Screen 24, 48  
 Split Screen 118  
 Move To Screen Edge 48  
 Scroll Other Window 118  
 Incremental Search 86  
 Reverse Incremental Search 87  
 String Search 88  
 Tags Search (m-X) Zmacs command 95  
 Global search and replace 94  
 Tags Tables and Search Domains 94  
 Searching 86  
 Searching for Appropriate Commands 35  
 Searching, Replacing, and Sorting 85  
 Select All Buffers As Tag Table (m-X) Zmacs command 94  
 Select Buffer 27, 105  
 SELECT C 30  
 Select Default Previous Buffer 105  
 SELECT E 30  
 SELECT E 10  
 SELECT F 30  
 SELECT I 30  
 SELECT key 30  
 SELECT L 30  
 SELECT M 30  
 SELECT P 30  
 Select Previous Buffer 105  
 Select System As Tag Table (m-X) Zmacs command 95  
 SELECT T 30  
 Select Tag Table (m-X) Zmacs command 94  
 SELECT X 30  
 Selected buffer 104  
 Selecting, Listing, and Examining Buffers 104  
 Self Insert and Complete 43, 44  
 Semicolon (;) comment indicator 157  
 Backward Sentence 23, 53

- Backward Kill Sentence 25, 72
- Cursor Motion by Sentence 52
- Forward Kill Sentence 23, 53
- Deleting Sentences 72
- Set Backspace (m-X) Zmacs command 128
- Set Base (m-X) Zmacs command 128
- Set commands 128
- Set commands for file and buffer attributes 127
- Set Comment Column 158
- Set Fill Column 180
- Set Fill Prefix 83
- Set Fonts (m-X) Zmacs command 128
- Dired Set Generation Retention Count 139
- Set Goal Column 57
- Set Key (m-X) Zmacs command 191
- Set Lowercase (m-X) Zmacs command 128
- Set Nofill (m-X) Zmacs command 128
- Set Package (m-X) Zmacs command 126, 127, 128
- Set Patch File (m-X) Zmacs command 128
- Set Pop Mark 76
- Set Tab Width (m-X) Zmacs command 128
- Set Variable 194
- Set Variable (m-X) Zmacs command 194
- Set Visited File Name (m-X) Zmacs command 110
- Set Vsp (m-X) Zmacs command 128
- zwei: \*set-attribute-update-list\* global variable 128
- Setting the Major Mode 143
- Settings 193
- How to Specify Zmacs Variable
- Backward Sexp 54
- Backward Kill Sexp 25, 69
- Forward Sexp 54
- Indent Sexp 151
- Kill Sexp 25, 69
- Mark Sexp 80
- Exchange Sexps 70
- Shift keys 5
- Electric Shift Lock mode 18
- Electric Shift Lock Mode (m-X) Zmacs command 180
- Two Windows Showing Region 117
- List the last sixty commands 36
- List the last sixty keystrokes 36
- Kill Some Buffers (m-X) Zmacs command 110
- Sort Lines (m-X) Zmacs command 100
- Sort Paragraphs (m-X) Zmacs command 100
- Sort Via Keyboard Macros 186
- Sort Via Keyboard Macros (m-X) Zmacs command 100
- Sorting 100
- Sorting 85
- Searching, Replacing, and Finding source code 156
- Editing the source code of a function 11
- Locating Source Code to Edit 166
- Source Compare 119
- Source Compare (m-X) Zmacs command 113
- Source Compare Merge 114
- Source Compare Merge (m-X) Zmacs command 113
- SPACE 13



Delete Horizontal	Space 26
	SPACE completion command 13
	SPACE Dired command 136
	SPACE Zmacs command 43
HELP	SPACE Zmacs command 12
Vertical	spacing 130
Move To	Specified Line 48
How to	Specify Zmacs Variable Settings 193
	Split Line 153
	Split Screen 118
Dired	Srccom 137
	Standard comtab 191
	Start Kbd Macro 183, 184
Erase backward to	start of line 71
	Start Patch (m-X) Zmacs command 172
	Start Private Patch (m-X) Zmacs command 172
Getting	Started 9
	Starting a New Line 20
	Starting Zmacs 10
Zmacs commands for finding out about the	state of buffers 201
Zmacs commands for finding out about the	state of Zmacs 201
* buffer	status indicator 19
Replace	String 90
Replace	String (m-X) Zmacs command 90
	String Search 88
Locating and Replacing	Strings Automatically 90
	Support buffers 98
Zmacs	support functions 98
	Swap Point And Mark 77
Select	System As Tag Table (m-X) Zmacs command 95
Entering File	System Editor 30
	System menu 30
Using the mouse to call up	System menu 30
R2:	System menu mouse click 50

T

T

T

SELECT	T 30
c-X	T Zmacs command 82
Insert	Tab 150
Set	Tab Width (m-X) Zmacs command 128
	TAB Zmacs command 150
	Tab-Width file attribute 130
Command	table 191
Find Files in Tag	Table (m-X) Zmacs command 97
Select All Buffers As Tag	Table (m-X) Zmacs command 94
Select System As Tag	Table (m-X) Zmacs command 95
Select Tag	Table (m-X) Zmacs command 94
Visit Tag	Table (m-X) Zmacs command 97
Command	tables 5
List Tag	Tables (m-X) Zmacs command 95
Tags	Tables and Search Domains 94
Find Files in	Tag Table (m-X) Zmacs command 97
Select	Tag Table (m-X) Zmacs command 94
Select All Buffers As	Tag Table (m-X) Zmacs command 94
Select System As	Tag Table (m-X) Zmacs command 95
Visit	Tag Table (m-X) Zmacs command 97
List	Tag Tables (m-X) Zmacs command 95

- Tags Multiple Query Replace (m-X) Zmacs command 96
  - Tags Query Replace (m-X) Zmacs command 95
  - Tags Search (m-X) Zmacs command 95
  - Tags Tables and Search Domains 94
  - Teco mode 145
  - Terminal 30
  - Text 112
  - Text 61
  - Text 25, 66
  - Text 20
  - text 80
  - text 33
  - text from register into buffer 78
  - Text mode 18, 144
  - thing mouse click 49
  - This Indentation 152
  - this manual 2
  - To Buffer 112
  - To File (m-X) Zmacs command 112
  - To File (m-X) Zmacs command 112
  - To Indentation 151
  - To Screen Edge 48
  - To Specified Line 48
  - Toggle 179
  - Top Level 32
  - Top-level expression 55
  - Top-Level Expressions 55
  - Trace (m-X) Zmacs command 38
  - Transposing Characters 66
  - Transposing Lines 71
  - Transposing Lisp Expressions 69
  - Transposing Text 61
  - Transposing Words 68
  - Trouble 32
  - Two window mode 117
  - Two Windows 117
  - Two Windows 117
  - Two Windows 117
  - Two Windows 117
  - Two Windows Showing Region 117
  - Typeout 15
  - Typeout window 15, 34
  - types 122
  - types and major modes 182
  - types of major modes 182
  - Typical use of Zmacs 156
  - Typos 20
- U**
- U Dired command 138
  - U Zmacs command 12, 36
  - Unbalanced Parentheses (m-X) Zmacs command 38
  - Unbalanced Parentheses When Saving 195
  - Uncomment Region (m-X) Zmacs command 159
  - Undelete 138
  - Undelete 138

- Undo all changes to buffer 109
- Undumped 140
- Unique 43
- Unknown attribute 126
- Up Comment Line 158
- Up List 55
- Up List 55
- Up List (c-m-X) Zmacs command 69
- Up Real Line 23, 56, 136
- up System menu 30
- Update Attribute List (m-X) Zmacs command 125
- Uppercase Code in Region (m-X) Zmacs command 148
- Uppercase Initial 148
- Uppercase Region 148
- Uppercase Word 148
- User-defined major modes 182

Dired Next

Complete And Exit If

Backward

Forward

Kill Backward

Using the mouse to call

V

V Dired command 137

V Zmacs command 193

V Zmacs command 193

V Zmacs command 107

V Zmacs command 12, 36, 193

variable 182

variable 128

Variable 194

Variable (m-X) Zmacs command 193

Variable (m-X) Zmacs command 194

Variable Apropos 193

Variable Apropos Zmacs command 193

Variable At Point 38

Variable Settings 193

variables 12, 36

Variables (m-X) Zmacs command 193

Verbose 160, 161

versions 137

Versions 138

versions 102

Vertical spacing 130

Via Keyboard Macros 186

Via Keyboard Macros (m-X) Zmacs command 100

View Buffer 107

View Buffer (m-X) Zmacs command 107

View File 137

View File (m-X) Zmacs command 120

View File Properties 120

View File Properties (m-X) Zmacs command 120

View Kbd Macro (m-X) Zmacs command 184

View Two Windows 117

Visit File 109

Visit Tag Table (m-X) Zmacs command 97

Visited File Name (m-X) Zmacs command 110

vs. Killing 62

Vsp (m-X) Zmacs command 128

c-HELP

c-m-?

c-X

HELP

**\*fs:file-type-mode-alist\*** Lisp

**ibase** global

Set

**zwei:\*major-mode-translations\*** Lisp

**zwei:\*set-attribute-update-list\*** global

Describe

Set

Describe

How to Specify Zmacs

Zmacs

List

Evaluate Region

Comparing file

Deleting Multiple

File

Sort

Sort

Dired

Set

Deleting

Set

V

V

Vsp file attribute 130

## W

## W

## W

	c-X	W Zmacs command	27
	HELP	W Zmacs command	12
	Compiler	Warnings (m-X) Zmacs command	163
	Edit Compiler	Warnings (m-X) Zmacs command	163
	Edit File	Warnings (m-X) Zmacs command	163
	Load Compiler	Warnings (m-X) Zmacs command	163
		Warnings about file attribute lists	126
		What is a Region?	74
Check Unbalanced Parentheses		When Saving	195
		Where Am I	37
	Fast	Where Am I	37
	Mark	Whole	81
Motion with Respect to the		Whole Buffer	60
	Set Tab	Width (m-X) Zmacs command	128
	Editor	Window	15
	Grow	Window	117
	One	Window	118
	Other	Window	118
	Recenter	Window	47
	Redisplaying the	Window	47
	Reposition	Window	48
Return to previous		window	30
	Scroll Other	Window	118
	Typeout	window	15, 34
	Positioning	Window Around Definition	48
		Window Commands	117
	One	Window Default	194
	Two	window mode	117
Modified Two		Windows	117
Multiple		windows	102
Two		Windows	117
Using Two		Windows	117
View Two		Windows	117
Two		Windows Showing Region	117
Cursor motion		within buffer	60
	Backward	Word	23, 52
	Backward Kill	Word	25, 68
Cursor Motion by		Word	52
	Forward	Word	23, 52
	Kill	Word	25, 68
Lowercase		Word	148
Mark		Word	80
Uppercase		Word	148
		Word Abbrev mode	18
		Word Abbrev Mode (m-X) Zmacs command	181
		Word Mode (m-X) Zmacs command	180
Deleting and Transposing		Words	68
	Exchange	Words	68
		Working With Buffers and Files	102
		Working with Regions	73
		Wraparound Lines	46
		Wrapping Lines	20
		Write File	27, 109
		Write File Zmacs command	29

X

X

X

SELECT X 30

Y

Y

Y

Yank 64  
c-Ø c-m-Y yank command 14  
c-m-Y yank command 14  
c-Ø c-Y yank command 13  
c-Y yank command 13  
m-Y yank command 13, 14, 64  
Yank Pop 64  
Yanking 13, 33  
Yanking in the command history 14  
Yanking in the kill history 13

Z

Z

Z

+ flag in Zmacs 103  
Entering Zmacs 10, 30  
Exiting Zmacs 30  
Getting Help in Zmacs 12  
Introduction to Zmacs 4  
Invoking Zmacs 10  
Leaving Zmacs 30  
Reinitializing Zmacs 11  
Starting Zmacs 10  
Typical use of Zmacs 156  
Using the mouse to enter Zmacs 10  
Zmacs commands for finding out about the state of Zmacs 201  
RUBOUT Zmacs character 66  
ABORT Zmacs command 32  
Abort Patch (m-X) Zmacs command 174  
Add Patch (m-X) Zmacs command 173  
Add Patch Changed Definitions (m-X) Zmacs command 173  
Add Patch Changed Definitions of Buffer (m-X) Zmacs command 173  
Append To File (m-X) Zmacs command 112  
Arglist (m-X) Zmacs command 37  
Atom Query Replace (m-X) Zmacs command 93  
Atom Word Mode (m-X) Zmacs command 180  
Auto Fill Lisp Comments Mode (c-m-X) Zmacs command 180  
Auto Fill Mode (m-X) Zmacs command 180  
c-¥ Zmacs command 90  
c-. Zmacs command 98, 168  
c-; Zmacs command 157  
c-< Zmacs command 81  
c-= Zmacs command 37  
c-> Zmacs command 81  
c-A Zmacs command 23, 57  
c-B Zmacs command 23, 52  
c-D Zmacs command 25, 33, 66  
c-E Zmacs command 23, 57  
c-F Zmacs command 23, 51  
c-G Zmacs command 32, 33  
c-HELP V Zmacs command 193  
c-K Zmacs command 25, 71  
c-L Zmacs command 47

c-m-)	Zmacs command	55
c-m-]	Zmacs command	56
c-m-(	Zmacs command	55
c-m-;	Zmacs command	157
c-m-? V	Zmacs command	193
c-m-@	Zmacs command	80
c-m-A	Zmacs command	55
c-m-B	Zmacs command	54
c-m-D	Zmacs command	54
c-m-E	Zmacs command	56
c-m-F	Zmacs command	54
c-m-H	Zmacs command	80
c-m-K	Zmacs command	25, 69
c-m-L	Zmacs command	105
c-m-M	Zmacs command	151
c-m-N	Zmacs command	53
c-m-O	Zmacs command	153
c-m-P	Zmacs command	53
c-m-Q	Zmacs command	151
c-m-R	Zmacs command	48
c-m-RETURN	Zmacs command	151
c-m-RUBOUT	Zmacs command	25, 69
c-m-sh-E	Zmacs command	160
c-m-SPACE	Zmacs command	76
c-m-T	Zmacs command	70
c-m-U	Zmacs command	55
c-m-V	Zmacs command	118
c-m-Z	Zmacs command	161
c-m-[	Zmacs command	55
c-m-\	Zmacs command	151
c-m-^	Zmacs command	152
c-N	Zmacs command	23, 56, 183
c-O	Zmacs command	153
c-P	Zmacs command	23, 56
c-R	Zmacs command	87
c-sh-A	Zmacs command	37
c-sh-C	Zmacs command	82
c-sh-D	Zmacs command	37
c-sh-E	Zmacs command	160
c-sh-M	Zmacs command	165
c-sh-V	Zmacs command	38
c-SPACE	Zmacs command	76
c-T	Zmacs command	66
c-U	Zmacs command	22
c-V	Zmacs command	24, 47
c-W	Zmacs command	25, 82
c-X	Zmacs command	86, 183
c-X 1	Zmacs command	118
c-X 2	Zmacs command	117
c-X 3	Zmacs command	117
c-X 4	Zmacs command	117
c-X 8	Zmacs command	117
c-X ;	Zmacs command	158
c-X =	Zmacs command	37
c-X A	Zmacs command	112
c-X B	Zmacs command	27, 105
c-X c-;	Zmacs command	159
c-X c-B	Zmacs command	103, 106

c-X c-D	Zmacs command	119
c-X c-F	Zmacs command	28, 29, 108
c-X c-G	Zmacs command	32
c-X c-I	Zmacs command	152
c-X c-L	Zmacs command	148
c-X c-m-L	Zmacs command	105
c-X c-m-SPACE	Zmacs command	76
c-X c-N	Zmacs command	57
c-X c-O	Zmacs command	26, 153
c-X c-P	Zmacs command	81
c-X c-S	Zmacs command	28, 109
c-X c-T	Zmacs command	71
c-X c-U	Zmacs command	148
c-X c-V	Zmacs command	109
c-X c-W	Zmacs command	29, 109
c-X c-X	Zmacs command	77
c-X D	Zmacs command	131
c-X E	Zmacs command	184
c-X F	Zmacs command	27, 180
c-X H	Zmacs command	81
c-X K	Zmacs command	110
c-X L	Zmacs command	38
c-X O	Zmacs command	118
c-X Q	Zmacs command	185
c-X RUBOUT	Zmacs command	25, 72
c-X S	Zmacs command	27
c-X T	Zmacs command	82
c-X V	Zmacs command	107
c-X W	Zmacs command	27
c-X ^	Zmacs command	117
c-Z	Zmacs command	30
Change File Properties (m-X)	Zmacs command	123
Clean Directory (m-X)	Zmacs command	123
CLEAR-INPUT	Zmacs command	71
Compile Buffer (m-X)	Zmacs command	162
Compile Changed Definitions (m-X)	Zmacs command	162
Compile Changed Definitions of Buffer (m-X)	Zmacs command	162
Compile File (m-X)	Zmacs command	162
Compile Region (m-X)	Zmacs command	82
Compiler Warnings (m-X)	Zmacs command	163
Copy File (m-X)	Zmacs command	121
Create Directory (m-X)	Zmacs command	123
Create Link (m-X)	Zmacs command	122
Decrypt Buffer (m-X)	Zmacs command	108
Deinstall Macro (m-X)	Zmacs command	186
Delete File (m-X)	Zmacs command	122
Describe Variable (m-X)	Zmacs command	193
Dired (m-X)	Zmacs command	131
Edit Buffers (m-X)	Zmacs command	107
Edit Callers (m-X)	Zmacs command	169
Edit Changed Definitions (m-X)	Zmacs command	167
Edit Changed Definitions of Buffer (m-X)	Zmacs command	168
Edit Compiler Warnings (m-X)	Zmacs command	163
Edit File Warnings (m-X)	Zmacs command	163
Electric Font Lock Mode (m-X)	Zmacs command	180
Electric Shift Lock Mode (m-X)	Zmacs command	180
EMACS Mode (m-X)	Zmacs command	181
Encrypt Buffer (m-X)	Zmacs command	108

End Kbd Macro	Zmacs command	183
Evaluate and Replace Into Buffer (m-X)	Zmacs command	92, 161
Evaluate Buffer (m-X)	Zmacs command	160
Evaluate Changed Definitions (m-X)	Zmacs command	161
Evaluate Changed Definitions of Buffer (m-X)	Zmacs command	161
Evaluate Into Buffer (m-X)	Zmacs command	160
Evaluate Region (m-X)	Zmacs command	160, 161
Find File	Zmacs command	28, 29
Find File In Fundamental Mode (m-X)	Zmacs command	110
Find Files in Tag Table (m-X)	Zmacs command	97
Find Unbalanced Parentheses (m-X)	Zmacs command	38
Finish Patch (m-X)	Zmacs command	174
Hardcopy Buffer (m-X)	Zmacs command	108
Hardcopy File (m-X)	Zmacs command	120
HELP ?	Zmacs command	12
HELP A	Zmacs command	12, 35
HELP C	Zmacs command	12, 34
HELP D	Zmacs command	12, 35
HELP L	Zmacs command	12, 36
HELP SPACE	Zmacs command	12
HELP U	Zmacs command	12, 36
HELP V	Zmacs command	12, 36, 193
HELP W	Zmacs command	12
Insert Buffer (m-X)	Zmacs command	112
Insert File (m-X)	Zmacs command	112
Install Command (m-X)	Zmacs command	192
Install Macro (m-X)	Zmacs command	186
Kill Backward Up List (c-m-X)	Zmacs command	69
Kill Or Save Buffers (m-X)	Zmacs command	110
Kill Some Buffers (m-X)	Zmacs command	110
LINE	Zmacs command	151
Lisp Mode (m-X)	Zmacs command	156
List Buffers	Zmacs command	103
List Callers (m-X)	Zmacs command	170
List Changed Definitions of Buffer (m-X)	Zmacs command	169
List Definitions (m-X)	Zmacs command	168
List Files (m-X)	Zmacs command	119
List Tag Tables (m-X)	Zmacs command	95
List Variables (m-X)	Zmacs command	193
Load Compiler Warnings (m-X)	Zmacs command	163
m-]	Zmacs command	23, 58
m- )	Zmacs command	56
m-^	Zmacs command	90
m-(	Zmacs command	164
m-.	Zmacs command	166
m-;	Zmacs command	157
m-<	Zmacs command	24, 60
m-=	Zmacs command	38
m->	Zmacs command	24, 60
m-@	Zmacs command	80
m-A	Zmacs command	23, 53
m-B	Zmacs command	23, 52
m-C	Zmacs command	148
m-D	Zmacs command	25, 68
m-E	Zmacs command	53
m-ESCAPE	Zmacs command	160
m-F	Zmacs command	23, 52
m-H	Zmacs command	80



m-K	Zmacs command	25, 72
m-L	Zmacs command	148
m-LINE	Zmacs command	158
m-M	Zmacs command	151
m-N	Zmacs command	158
m-O	Zmacs command	152
m-P	Zmacs command	158
m-R	Zmacs command	48
m-RETURN	Zmacs command	151
m-RUBOUT	Zmacs command	25, 68
m-S	Zmacs command	150
m-sh-C	Zmacs command	162
m-sh-D	Zmacs command	37
m-sh-E	Zmacs command	161
m-T	Zmacs command	68
m-U	Zmacs command	148
m-V	Zmacs command	24, 48
m-W	Zmacs command	82
m-Z	Zmacs command	162
m-[	Zmacs command	23, 58
m-\	Zmacs command	26
m-^	Zmacs command	26, 152
Macro Expand Expression (m-X)	Zmacs command	165
Multiple Edit Callers (m-X)	Zmacs command	170
Multiple List Callers (m-X)	Zmacs command	171
Multiple Query Replace (m-X)	Zmacs command	91
Multiple Query Replace From Buffer (m-X)	Zmacs command	92
Name Last Kbd Macro (m-X)	Zmacs command	185
Overwrite Mode (m-X)	Zmacs command	181
Prepend To File (m-X)	Zmacs command	112
Query Replace (m-X)	Zmacs command	90
Query Replace Last Kill (m-X)	Zmacs command	92
Query Replace Let Binding (m-X)	Zmacs command	92
Reap File (m-X)	Zmacs command	122
Rename Buffer (m-X)	Zmacs command	108
Rename File (m-X)	Zmacs command	120
Reparsed Attribute List (m-X)	Zmacs command	125
Replace String (m-X)	Zmacs command	90
RETURN	Zmacs command	43
Revert Buffer (m-X)	Zmacs command	109
RUBOUT	Zmacs command	25, 33
Save All Files (m-X)	Zmacs command	108
Save File	Zmacs command	28
Select All Buffers As Tag Table (m-X)	Zmacs command	94
Select System As Tag Table (m-X)	Zmacs command	95
Select Tag Table (m-X)	Zmacs command	94
Set Backspace (m-X)	Zmacs command	128
Set Base (m-X)	Zmacs command	128
Set Fonts (m-X)	Zmacs command	128
Set Key (m-X)	Zmacs command	191
Set Lowercase (m-X)	Zmacs command	128
Set Nofill (m-X)	Zmacs command	128
Set Package (m-X)	Zmacs command	126, 127, 128
Set Patch File (m-X)	Zmacs command	128
Set Tab Width (m-X)	Zmacs command	128
Set Variable (m-X)	Zmacs command	194
Set Visited File Name (m-X)	Zmacs command	110
Set Vsp (m-X)	Zmacs command	128

Sort Lines (m-X)	Zmacs command	100
Sort Paragraphs (m-X)	Zmacs command	100
Sort Via Keyboard Macros (m-X)	Zmacs command	100
Source Compare (m-X)	Zmacs command	113
Source Compare Merge (m-X)	Zmacs command	113
SPACE	Zmacs command	43
Start Patch (m-X)	Zmacs command	172
Start Private Patch (m-X)	Zmacs command	172
TAB	Zmacs command	150
Tags Multiple Query Replace (m-X)	Zmacs command	96
Tags Query Replace (m-X)	Zmacs command	95
Tags Search (m-X)	Zmacs command	95
Trace (m-X)	Zmacs command	38
Uncomment Region (m-X)	Zmacs command	159
Update Attribute List (m-X)	Zmacs command	125
Uppercase Code in Region (m-X)	Zmacs command	148
Variable Apropos	Zmacs command	193
View Buffer (m-X)	Zmacs command	107
View File (m-X)	Zmacs command	120
View File Properties (m-X)	Zmacs command	120
View Kbd Macro (m-X)	Zmacs command	184
Visit Tag Table (m-X)	Zmacs command	97
Word Abbrev Mode (m-X)	Zmacs command	181
Write File	Zmacs command	29
Finding Out About	Zmacs Commands	34
Mouse-sensitive	Zmacs commands	50
	Zmacs commands for finding out about flavors	202
	Zmacs commands for finding out about Lisp	201
	Zmacs commands for finding out about the state of buffers	201
	Zmacs commands for finding out about the state of Zmacs	201
	Zmacs commands for interacting with Lisp	202
Customizing the	Zmacs Environment	177
	Zmacs support functions	98
How to Specify	Zmacs Variable Settings	193
	Zmacs variables	12, 36
Entering	Zmail	30
	<b>zwei:*major-mode-translations*</b> Lisp variable	182
	<b>zwei:*set-attribute-update-list*</b> global variable	128
	<b>zwei:defmajor</b>	182
	<b>zwei:edit-functions</b> function	11

*symbolics*<sup>TM</sup>

---

# **DEBUG** Debugger

---

---

Cambridge, Massachusetts

---

# **Debugger**

**# 990015**

**February 1984**

**This document corresponds to Release 5.0.**

This document was prepared by the Documentation Group of Symbolics, Inc.

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Symbolics is a trademark of Symbolics, Inc., Cambridge, Massachusetts.

Copyright © 1981, 1979, 1978 Massachusetts Institute of Technology.  
All rights reserved.

Enhancements copyright © 1984, 1983, 1982 Symbolics, Inc. of Cambridge,  
Massachusetts.

All rights reserved. Printed in USA.

This document may not be reproduced in whole or in part without the prior written consent of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

## Table of Contents

	Page
<b>1. Entering the Debugger</b>	<b>1</b>
1.1 Error Display	2
<b>2. How to Use the Debugger</b>	<b>5</b>
<b>3. Debugger Commands</b>	<b>9</b>
<b>4. Summary of Debugger Commands</b>	<b>15</b>
<b>5. Summary of Debugging Aids</b>	<b>17</b>
<b>6. Tracing Function Execution</b>	<b>19</b>
6.1 Options to trace	20
6.2 Controlling the Format of trace Output	23
6.3 Untracing Function Execution	24
<b>7. Advising a Function</b>	<b>25</b>
7.1 Designing the Advice	27
7.2 :around Advice	28
7.3 Advising One Function Within Another	29
<b>8. Stepping Through an Evaluation</b>	<b>31</b>
<b>9. evalhook</b>	<b>33</b>
9.1 applyhook	35
<b>10. The MAR</b>	<b>37</b>
<b>11. Variable Monitoring</b>	<b>39</b>
<b>Index</b>	<b>41</b>

## 1. Entering the Debugger

When an error condition is signalled and no handlers decide to handle the error, an interactive Debugger is entered to allow you to look around and see what went wrong and to help you continue the program or abort it. This section describes how to use the Debugger and the various debugging facilities.

The Debugger is invoked automatically when errors arise during program execution or when you explicitly cause an error, for example, by typing a nonsense symbol name, such as **ahsdgf**, at the Lisp read-eval-print loop.

You can also enter the Debugger explicitly by pressing **m-SUSPEND**. Adding the **CONTROL** modifier to this combination has the effect of saying "enter the Debugger immediately". Thus, you can:

- Press **m-SUSPEND** while the currently running program or read-eval-print loop is reading from the console.
- Press **c-m-SUSPEND** so that the currently running program enters the Debugger whether or not it is reading from the console.

*Note:* Pressing the **SUSPEND** key without the **META** modifier or just pressing **c-SUSPEND** enters a read-eval-print loop rather than the Debugger.

You can use the **dbg** function in your source code to help detect errors in your programs.

- Insert a call to **dbg** (with no arguments) into your code and then recompile.
- Call **dbg** with an argument of *process* to force a process into the Debugger.

### **dbg** &optional *process*

*Function*

Forces *process* into the Debugger so that you can look at its current state. **dbg** sets up a restart handler for **c-Z**, **ABORT**, and **RESUME** that exits from the **dbg** function back to the original process. The message for this restart handler is "Allow process to continue". You can use **c-T**, **c-R**, **c-m-R**, and other similar Debugger commands when you enter the Debugger via **dbg**.

- With no argument, it enters the Debugger as if an error had occurred for the current process. It is not an error; in particular, **errset** and **catch-error** do not handle it. You can include this form in program source code as a means of entering the Debugger. This is useful for breakpoints and causes a special compiler warning.
- With an argument of **t** (rather than a process, window, or stack group), it finds a process that has sent an error notification.

Suppose you are running in process *X* and you use **dbg** on some process *Y*. Process *Y* is forced into the Debugger, no matter what it is doing.

Technically, it is "interrupted", similar to how `c-SUSPEND`, `c-ABORT` and `c-M-SUSPEND` work. Process Y starts running the Debugger, using the stream `debug-io`. `debug-io` gets the same stream as was bound to `terminal-io` in Process X. At this time, Process X waits in a state called `DBG` until Process Y leaves the Debugger, and so Process X does not contend for the stream.

For more information: See the special form `break`. See the section "Breakpoints".

## 1.1 Error Display

Errors are signalled by the Lisp Machine's microcode and by Lisp programs (by using `error` or related functions). Here is an example of an error:

```
foo
>>Trap: The variable FOO is unbound.
SI:*EVAL:
  Arg 0 (FORM): FOO
s-A, <RIGHT>:  Supply a value to use this time as the value of FOO
s-B, n-C:     Supply a value to store permanently as the value of FOO
s-C:         Retry the SYMEVAL instruction
s-D, <LEFT>:   Return to Lisp Top Level in Lisp Listener 1
*
```

>> indicates entry to the Debugger. The word immediately following >> shows what caused you to enter the Debugger; most commonly you see `Trap`, `Error`, or `Break`.

`Trap` indicates a microcode error.

`Error` indicates a software error.

`Break` indicates entry by keystroke or the `dbg` function.

The message that follows describes the error in English, in this example, an unbound variable. The next two lines in the example show the stack frame in which the error occurred — the function that was being called and the current value(s) of its argument(s).

The right-facing arrow (`→`) indicates that the Debugger is waiting for a command. Multiple arrow prompts signal recursive invocations of the Debugger.

The Debugger provides options for proceeding from the error or restarting from some prior point. When the Debugger is entered, all *proceed types*, *special commands*, or *restart handlers* available in the error context are assigned to keystrokes with the `SUPER` modifier, starting with `s-A`, `s-B`, and so on, from the most recently established (innermost) to the oldest (outermost). Also, the `RESUME` key is assigned to the innermost proceed type (or restart handler if there are no proceed types), and the `ABORT` key is assigned to the innermost restart handler. All these keystroke

assignments are displayed when you enter the Debugger or when you type the `c-l` Debugger command. (See the document *Signalling and Handling Conditions*.)

You can use one of these options or any of the Debugger commands. See the section "Debugger Commands". See the section "How to Use the Debugger". For details on the Debugger command keys: See the section "Special Keys".

Optionally, you can request that backtrace information appear when you enter the Debugger by setting the variable `dbg:*show-backtrace*` in your init file. See the section "Debugger Commands".





## 2. How to Use the Debugger

Once inside the Debugger, you can give a wide variety of commands. With these commands, you can see the arguments for the current stack frame, disassemble its code, return a value for the stack frame, move up and down the stack, and enter the editor to edit function definitions. Press the HELP key or the ? key to display a brief help message or c-HELP for documentation on all of the Debugger commands.

This section describes how to give the commands, and then explains them in approximate order of usefulness.

When the Debugger prompts you with →, you can do one of the following:

- Type a Lisp expression
- Type a Debugger command
- Use the input editor to recall a previous Lisp expression

The Debugger considers most keys used with a modifier (such as CONTROL or SUPER) to be commands. Most unmodified keys begin a Lisp expression; however, a few keys are commands even without a modifier. (See the section "Debugger Commands".)

The Debugger and the input editor use some of the same keys for commands. You can enter the input editor at any time by pressing a key that is not a Debugger command, for example, SPACE. Once there, you can type an input editor command that is also a Debugger command.

When you press a key that is not a command, the Debugger prompts with Eval:, which means that it will evaluate any Lisp expression that you type. The Debugger interprets the Lisp expression as a Lisp form and evaluates it in the context of the function that got the error. That is, all bindings that were in effect at the time of the error will be in effect when your form is evaluated, with certain exceptions explained later in this section. The result of the evaluation is printed, and the Debugger prompts again with an arrow.

If, during the typing of the form, you change your mind and want to get back to the Debugger's command level, press ABORT or c-G; the Debugger responds with an arrow prompt. In fact, you can press ABORT or c-G whenever the Debugger expects typein in order to flush what you are typing and get back to command level.

If a nontrivial error occurs in the evaluation of the Lisp expression, you are thrown into a second Debugger looking at the new error. The Debugger prompts with two arrows (↔) to show that you are inside two Debuggers. You can abort the computation and get back to the first Debugger by pressing the ABORT key. However, if the error is trivial the abort is done automatically and the original error message is reprinted.

Various Debugger commands ask for Lisp objects, such as an object to return or the name of a catch-tag. Whenever it requests a Lisp object, it expects you to type in a form; it will evaluate what you type in. This provides greater generality, since there are objects to which you might want to refer that cannot be typed, such as arrays. If the form you type is nontrivial (not just a constant form), the Debugger shows you the result of the evaluation and asks you if it is what you intended. It expects a Y or N answer. (See the function **y-or-n-p**.) If you answer negatively it asks you for another form. To exit the command, just press ABORT or c-G.

When the Debugger evaluates a form, the variable bindings at the point of error are in effect with the following exceptions:

- **terminal-io** is rebound to the stream the Debugger is using. **dbg:old-terminal-io** is bound to the value that **terminal-io** had at the point of error.
- **standard-input** and **standard-output** are rebound to be synonymous with **terminal-io**; their old bindings are saved in **dbg:old-standard-input** and **dbg:old-standard-output**.
- **query-io**, **debug-io**, and **error-output** are rebound to be synonymous with **terminal-io**; their old bindings are not directly accessible.
- **+** and **\*** are rebound to the Debugger's previous form and previous value. When the Debugger is first entered, **+** is the last form typed, which is typically the one that caused error, and **\*** is the value of the *previous* form. **++**, **+++**, **\*\***, **\*\*\***, **-**, and **//** are treated in an analogous fashion. See the section "The Lisp Top Level". When the Debugger is exited, all of these variables are restored to their original values; the interactions with the Debugger's read-eval-print loop do not affect the interactions with the top-level Lisp read-eval-print loop.
- **rubout-handler** and **read-preserve-delimiters** are rebound to **nil**, in case the error occurred while in the input editor or the reader.
- **evalhook** is rebound to **nil**, turning off the **step** facility if it had been in use when the error occurred. See the section "**evalhook**".
- **dbg:\*bound-handlers\*** and **dbg:\*default-handlers\*** are rebound to **nil**, preventing conditions signalled by the form the Debugger is evaluating from reaching condition handlers in the program being debugged. This prevents you from accidentally being thrown out of the Debugger.

Note that the variable bindings are those in effect at the point of error, *not* those of the current frame being examined.

The **m-S** command can be used to evaluate a special variable in the context of the

current frame. This works even for the special variables listed as exceptions (earlier in this section).



### 3. Debugger Commands

All Debugger commands are single characters, usually with the CONTROL or META modifiers. The single most useful command is ABORT (or c-Z), which exits from the Debugger and throws out of the computation that got the error. Often you are not interested in using the Debugger at all and just want to get back to the command level in the program you are running; ABORT lets you do this in one character.

The ABORT command returns control to the most recently established restart handler, usually a command or read-eval-print loop. Pressing ABORT multiple times throws you back to successively older read-eval-print or command loops until top level is reached. Pressing c-M-ABORT, on the other hand, always throws you to top level. (Note: c-M-ABORT is not a Debugger command but a system command, which is available from every program.)

Pressing ABORT in the middle of typing a form to be evaluated by the Debugger aborts that form and returns to the Debugger's command level, whereas pressing ABORT as a Debugger command returns out of the Debugger and the erring program to the *previous* command level.

Documentation is provided by the HELP or ? command, which types out a very brief explanation of the Debugger. The c-HELP command gives documentation for all of the Debugger commands. If you type c-L or press REFRESH, the Debugger clears the screen, redisplayes the error message and the current stack frame, displays a brief backtrace, and lists the special commands that apply to the particular error currently being handled and gives a one-line explanation of each of them.

Often you want to try to proceed from the error. To do this, use the RESUME (or c-C) command. The exact way RESUME works depends on the kind of error that happened. For some errors, there is no standard way to proceed, and RESUME just tells you so and returns to the Debugger's command level. For the very common "unbound variable" error, it requests that you supply the Lisp object that should be used in place of the (nonexistent) value of the symbol. For unbound-variable or undefined-function errors, you can also just type Lisp forms to set the variable or define the function, and then press RESUME; execution proceeds after the Debugger asks you to confirm that the new value is acceptable.

The Debugger knows about a *current stack frame* and has several commands that use it. The initially current stack frame is the one that signalled the error: either the one that got the microcode-detected error, or the one that called **ferror**, **error**, or a related function. When the Debugger starts up it shows you this frame in the following format:

```
FOO
  Arg 0 (X): 13
  Arg 1 (Y): 1
```

This means that **foo** was called with two arguments, whose names (in the Lisp source code) are **x** and **y**. The current values of **x** and **y** are **13** and **1** respectively. On the LM-2 these might not be the original arguments if the function happens to **setq** its argument variables. On the 3600, the Debugger shows the original arguments.

The Debugger provides several commands to allow you to examine the Lisp control stack and to make other frames current than the one that got the error. The control stack (or *regular pdl*) keeps a record of all functions that are currently active. If you call **foo** at Lisp's top level, and it calls **bar**, which in turn calls **baz**, and **baz** gets an error, then a *backtrace* (a backwards trace of the stack) would show all of this information.

The Debugger has three backtrace commands:

```
c-B
m-B
c-m-B
```

**c-B** simply displays the names of the functions on the stack, starting from the current frame; in the above example it would display

```
BAZ ← BAR ← FOO ← SI:*EVAL ← SI:LISP-TOP-LEVEL1 ← SI:LISP-TOP-LEVEL
```

The arrows indicate the direction of calling. A numeric argument specifies how many frames to display.

The **m-B** command displays a more extensive backtrace, indicating the names of the arguments to the functions and their current values; for the example above it might look like:

```
BAZ:
  Arg 0 (X): 13
  Arg 1 (Y): 1

BAR:
  Arg 0 (ADDEND): 13

FOO:
  Arg 0 (FROB): (A B C . D)
```

**c-m-B** is comparable to **m-B** but also includes internal frames of the Lisp interpreter, which normally are skipped.

The **c-N** command moves down to the next frame (that is, it changes the current frame to be the frame that called it) and displays the frame in this same format. **c-P** or **RETURN** moves up to the previous frame (that is, the one that this one called) and displays the frame in the same format.

**m-<** moves to the stack frame where the error occurred (the top or most recent frame), whereas **m->** goes to the bottom (the oldest frame); both display the new current frame. Use **c-P** after **m-<** to go up through **signal**, handlers, and so forth, in turn, until you get to the highest possible frame — the call to the Debugger itself.

**c-S** asks you for a string, and searches the stack for a frame whose executing function's name contains that string. That frame becomes current and is displayed.

**m-L** displays the current frame in full-screen format, which shows the arguments and their values, the local variables and their values, and the machine code with an arrow pointing to the next instruction to be executed. On the 3600, if a function **setq**s one of its arguments, **m-L** shows both the original argument supplied by the caller and the current value of the variable.

**m-N** moves to the next frame and displays it in full-screen format, and **m-P** moves to the previous frame and displays it in full-screen format.

Commands such as **c-N** and **m-N**, which are meaningful to repeat, take a prefix numeric argument and repeat that many times. The numeric argument is typed by using **c-** or **m-** and the number keys, as in the editor.

**c-E** puts you into the editor, looking at the source code for the function in the current frame. This is useful when you have found a function that caused the error and needs to be fixed. The editor command **c-Z** will return to the Debugger, if it is still there.

**m-C** is only available for such errors as an unbound variable or undefined function. It is similar to **c-C**, but actually **setq**s the variable or defines the function, so that the error does not happen again. **c-C** (or **RESUME**) provides a replacement value but does not actually change the variable.

**c-sh-P** is only available for such errors as an unbound variable or undefined function when there is a variable or function in another package that has the same name. It permits easy recovery when you forget to supply a package prefix.

**c-R** is used to return a value or values from the current frame; the frame that called that frame continues running as if the function of the current frame had returned. This command asks for as many values as the caller expects, which might be no values, one value, more than one, or an indefinite number of values. For each value it prompts you for a form, which it evaluates; it returns the resulting value, possibly after confirming it with you. If no values are expected, it requests confirmation before returning.

The **c-T** command does a **\*throw** to a given tag with a given value; you are prompted for the tag and the value.

**c-m-R** is a variation of **c-R**; it starts the current frame over with the same function and arguments. If the function has been redefined in the meantime (perhaps you edited it and fixed its bug) the new definition is used. **c-m-R** asks for confirmation before doing it.

The **c-m-N**, **c-m-P**, and **c-m-B** commands are like the corresponding **c-** commands but do not censor the stack. When running interpreted code, the Debugger tries to skip over frames that belong to functions of the interpreter, such as **\*eval**, **prog**, and **cond**, and only show "interesting" functions. The **c-m-U** command goes down the stack to the next interesting function and makes that the current frame.



`c-m-A` takes a numeric argument  $n$ , and displays the value of the  $n$ th argument of the current frame. The default value for the argument is 0, meaning the first frame. It leaves `*` set to the value of the argument, so that you can use the Lisp read-eval-print loop to examine it. It also leaves `+` set to a locative pointing to the argument on the stack, so that you can change that argument (by calling `rplacd` on the locative).

`c-m-L` is similar to `c-m-A`, but refers to the  $n$ th local variable of the frame.

`c-m-V` is similar to `c-m-A`, but refers to the  $n$ th value being returned by the frame. If the frame is not in the process of returning values, the command displays an error message. `c-m-V` is meaningful only when you are using `trap-on-exit` (see `c-X`) and looking at a frame that is about to return.

`c-m-F` is similar to `c-m-A`, but refers to the function executing in the frame; it ignores its numeric argument and does not allow you to change the function.

`c-m-H` describes any condition handlers established by the current frame (or its subframes if it is an interpreted function).

`c-m-S` describes any special-variable bindings in the current frame (or its subframes if it is an interpreted function).

`m-S` asks for the name of a special variable and displays its value in the binding context of the current frame. It leaves `*` set to the value that was displayed.

`m-I` (for "Instance") helps you examine the values of instance variables in the stack group being debugged. The command prompts you for the name of an instance variable and displays the value of that instance variable, inside the instance that is the value of `self` in the environment of the current frame.

`c-m-W` calls the Display Debugger, a window-oriented Debugger, which is not documented in this manual. It should, however, be usable without further documentation.

`c-M` sends a bug report. It creates a new process and runs the `bug` function in that process. It starts up a mail-sending window that contains a copy of the error message and an extensive backtrace of the stack. You are expected to supply context information explaining what you were doing when the problem occurred, preferably including a way for the person reading the bug report to make it happen again. The stack trace by itself is not adequate information for debugging. When you type the `END` key the bug report is transmitted as mail and the window containing the Debugger is reselected. You can also use normal window-switching commands such as `FUNCTION S` to switch back and forth between the Debugger and the mail-sending window while composing the bug report. A numeric argument to `c-M` controls the number of stack frames in the backtrace that have complete information. The current stack frame at the time `c-M` is typed begins the backtrace, so you might want to type `m-<` before `c-M` if you have been examining other frames than the one that got the error.

**c-x** toggles the trap-on-exit flag of the current frame and displays its new state. **m-x** sets the trap-on-exit flag in the current frame and all its callers. **c-m-x** clears the trap-on-exit flag in the current frame and all its callers. If a frame with the trap-on-exit flag set returns or is thrown through, the Debugger is entered. Press **RESUME** to continue returning or throwing. The **ABORT** key, however, bypasses the trap-on-exit mechanism.

The Debugger's command loop lets you type in Lisp forms, which it reads, evaluates, and prints. When you are typing these forms, you can use the following functions to examine or modify the arguments, locals, function object, and values being returned in the current frame.

**dbg:arg** *name-or-number* *Function*  
 Returns the value of argument *name-or-number* in the current stack frame. **(setf (dbg:arg *n*) *x*)** sets the value of the argument *n* in the current frame to the value of *x*. *name-or-number* can be the number of the argument (for example, 0 to specify the first argument) or the name of the argument. This function can be called only from the read-eval-print loop of the Debugger.

**dbg:loc** *name-or-number* *Function*  
 Returns the value of the local variable *name-or-number* in the current stack frame. **(setf (dbg:loc *n*) *x*)** sets the value of the local variable *n* in the current frame to the value of *x*. *name-or-number* can be the number of the local variable (for example, 0 to specify the first local variable) or the name of the local variable. This function can be called only from the read-eval-print loop of the Debugger. *(dbg:loc 'epsilon)*

**dbg:fun** *Function*  
 Returns the function object of the current stack frame. **(setf (dbg:fun) *x*)** sets the function object of the current frame to the value of *x*. This function can be called only from the read-eval-print loop of the Debugger.

**dbg:val** &optional *val-no* 0 *Function*  
 Returns the value of the *val-no*th value to be returned from the current stack frame. **(setf (dbg:val *val-no*) *x*)** sets the value of the *val-no*th value to be returned from the current frame to the value of *x*. *val-no* must be a fixnum (since values do not have names) and defaults to 0. **(dbg:val)** without a value number gives the first value. This function can be called only from the read-eval-print loop of the Debugger. *(dbg)*

The Debugger uses the following variables:

**dbg:\*frame\*** *Variable*  
 Inside the read-eval-print loop of the Debugger, the value of **dbg:\*frame\*** is the location of the current frame.

**dbg:\*defer-package-dwim\****Variable*

When this is **nil** (the default), the Debugger searches over all packages to find any look-alike symbols, when errors concerning unbound variables occur.

When the option is not **nil**, the search does not occur until you type **c-sh-P**. In this case the Debugger offers **c-sh-P** in the list of commands even if the search would find no look-alike symbols.

**dbg:\*debug-io-override\****Variable*

If the value of this variable is **nil** (the default), the Debugger uses the stream that is the value of **debug-io**. But if the value of **dbg:\*debug-io-override\*** is not **nil**, the Debugger uses the stream that is the value of this variable instead. This variable should always be set (using **setq**), not bound, so all processes and stack groups can see it.

**dbg:\*show-backtrace\****Variable*

Backtrace information appears when you enter the Debugger. The default is **nil**. `@symindexm(pkg={dbg:},sym={*show-backtrace*},key={show-backtrace*})`

*Value**Meaning***nil**

The Debugger startup message does not include any backtrace information.

**t**

The Debugger startup message includes a three-element backtrace.

## 4. Summary of Debugger Commands

c-A	Displays argument list of function in current frame. It displays only the names of the arguments, not their values.
c-M-A	Examines or changes the $n$ th argument of the current frame.
c-B	Displays a brief backtrace, including only the names of the functions.
m-B	Displays a more extensive backtrace than c-B, including the names of the arguments to the functions and their current values.
c-M-B	Displays a longer backtrace than c-B and m-B, providing the names of the arguments to the functions and their current values as well as the internal frames of the Lisp interpreter.
c-C, RESUME	Attempts to continue execution, if possible.
m-C	Attempts to continue, <b>setting</b> the unbound variable or otherwise permanently fixing the error.
c-E	Puts you in the editor with the cursor positioned at the source code for the function in the current frame. <i>c-Z to return</i>
c-M-F	Sets * to the function in the current frame.
c-G or ABORT	Quits various Debugger commands; use to escape from typing in a form.
c-M-H	Describes any condition handlers established by the current frame.
m-I	Evaluates an instance variable of the instance that is <b>self</b> in the current frame.
c-L, REFRESH	Redisplays error message and current frame.
* m-L	Displays full-screen typeout of current frame.
c-M-L	Gets local variable $n$ .
c-M	Sends mail to report a bug.
c-N, LINE	Moves to next frame. With argument of $n$ , moves down $n$ frames.
m-N	Moves to next frame with full-screen typeout. With argument of $n$ , moves down $n$ frames.
c-M-N	Moves to next frame even if it is "uninteresting". With argument of $n$ , moves down $n$ frames.
c-P, RETURN	Moves to previous frame. With argument of $n$ , moves up $n$ frames.
m-P	Moves to previous frame with full-screen typeout. With argument of $n$ , moves up $n$ frames.

c-m-P	Moves to previous frame even if it is "uninteresting". With argument of $n$ , moves up $n$ frames.
c-R	Returns from the current frame.
c-m-R	Reinvokes the function in the current frame (throws back to it and starts it over at its beginning).
c-S	Searches for a frame containing a user-specified function.
m-S	Evaluates a special variable in the binding context of the current frame.
c-m-S	Describes any special-variable bindings established by the current frame.
c-T	Throws a value to a tag.
c-m-U	Moves down the stack to the next "interesting" frame.
c-m-V	Gets the $n$ th value being returned by the current frame.
c-m-W	Invokes the Display Debugger.
c-X	Toggles the trap-on-exit flag of the current frame.
m-X	Sets the trap-on-exit flag in the current frame and all its callers.
c-m-X	Clears the trap-on-exit flag in the current frame and all its callers.
c-Z, ABORT	Aborts the computation and throws back to the most recent <b>break</b> or Debugger, to the program's "command level", or to Lisp top level.
? or HELP	Displays a brief help message.
c-HELP	Displays a detailed help message.
m-<	Goes to top or most recent frame of stack, the stack where the error occurred.
m->	Goes to bottom or oldest frame of stack.
c-0—c-m-9	Numeric arguments to the following command are specified by typing a decimal number with the CONTROL and/or META keys held down.

## 5. Summary of Debugging Aids

Anyone who writes programs for the Lisp Machine should become familiar with these debugging facilities.

- The *trace* facility provides the ability to perform certain actions at the time a function is called or at the time it returns. The actions may be simple typeout, or more sophisticated debugging functions. See the section "Tracing Function Execution".
- The *advise* facility is a somewhat similar facility for modifying the behavior of a function. See the section "Advising a Function".
- The *step* facility allows the evaluation of a form to be intercepted at every step so that the user may examine just what is happening throughout the execution of the form. See the section "Stepping Through an Evaluation".
- The *evalhook* facility allows you to get at a particular Lisp form whenever the evaluator is called. The step facility uses **evalhook**. See the section "**evalhook**".
- The *MAR* facility (available only on the LM-2) provides the ability to cause a trap on any memory reference to a word (or a set of words) in memory. If something is getting clobbered by agents unknown, the MAR facility can help track down the source of the problem.

See the section "Tracing and Stepping".



## 6. Tracing Function Execution

The trace facility allows you to *trace* some functions. Tracing is useful when you need to find out why a program behaves in an unexpected manner, particularly when you suspect that arguments are being passed incorrectly or functions are being called in the wrong sequence. The trace facility is closely compatible with Maclisp.

Certain special actions are taken when a traced function is called and when it returns. The default tracing action prints a message when the function is called, showing its name and arguments, and another message when the function returns, showing its name and value(s). See the section "Tracing".

You invoke the trace facility in several ways:

- Use the **trace** and **untrace** special forms.
- Click on [Trace] in the System menu. Enter or point to the function to be traced; a menu of options pops up.
- Invoke the Trace (M-X) command in the editor. Enter the function to be traced; a menu of options pops up.

The menu options are also available with **trace**; however, the syntax is complex. For a table explaining the correspondence between menu options and **trace** options: See the section "Tracing".

### **trace**

*Special Form*

A **trace** form looks like:

```
(trace spec-1 spec-2 ...)
```

Each *spec* can take any of the following forms:

a symbol

This is a function name, with no options. The function is traced in the default way, printing a message each time it is called and each time it returns.

a list (*function-name option-1 option-2 ...*)

*function-name* is a symbol and the *options* control how it is to be traced. For a list of the various options: See the section "Options to **trace**". Some options take arguments, which should be given immediately following the option name.

a list (**:function** *function-spec option-1 option-2 ...*)

This option is like the previous form except that *function-spec* need not be a symbol. (See the section "Function Specs".) It exists because if *function-name* were a list in the previous form, it would instead be interpreted as the following form:



a list ((*function-1 function-2...*) *option-1 option-2 ...*)

All of the functions are traced with the same options. Each *function* can be either a symbol or a general function-spec.

**trace** returns as its value a list of names of all functions it traced. If called with no arguments, as just (**trace**), it returns a list of all the functions currently being traced.

If you attempt to trace a function already being traced, **trace** calls **untrace** before setting up the new trace.

Tracing is implemented with encapsulation, so if the function is redefined (for example, with **defun** or by loading it from a compiled code file) the tracing is transferred from the old definition to the new definition.

See the section "Encapsulations".

## 6.1 Options to trace

The following **trace** options exist:

### **:break** *pred*

Enters a breakpoint after printing the entry trace information but before applying the traced function to its arguments, if and only if *pred* evaluates to non-**nil**. During the breakpoint, the symbol **arglist** is bound to a list of the arguments of the function.

### **:exitbreak** *pred*

This is just like **:break** except that the breakpoint is entered after the function has been executed and the exit trace information has been printed, but before control returns. During the breakpoint, the symbol **arglist** is bound to a list of the arguments of the function, and the symbol **values** is bound to a list of the values that the function is returning.

**:error** Calls the Debugger when the function is entered. Use **RESUME** (or **c-C**) to continue execution of the function. If this option is specified, no printed trace output appears other than the error message displayed by the Debugger. (Note: If you also want to call the Debugger when the function returns, use the Debugger's **c-X** command.)

**:step** Steps through the function whenever it is called. See the section "Stepping Through an Evaluation".

### **:entrycond** *pred*

Prints trace information on function entry only if *pred* evaluates to non-**nil**.

### **:exitcond** *pred*

Prints trace information on function exit only if *pred* evaluates to non-**nil**.

**:cond** *pred*

Prints trace information on function entry and exit only if *pred* evaluates to non-**nil**.

**:wherein** *function*

Traces the function only when it is called, directly or indirectly, from the specified function *function*. You can give several trace specs to **trace**, all specifying the same function but with different **:wherein** options, so that the function is traced in different ways when called from different functions.

This is different from **advise-within**, which only affects the function being advised when it is called directly from the other function. The **trace :wherein** option means that when the traced function is called, the special tracing actions occur if the other function is the caller of this function, or its caller's caller, or its caller's caller's caller, and so on.

**:argpdl** *pdl*

Specifies a symbol *pdl*, whose value is initially set to **nil** by **trace**. When the function is traced, a list of the current recursion level for the function, the function's name, and a list of arguments is pushed onto the *pdl* when the function is entered, and then popped when the function is exited. The *pdl* can be inspected from within a breakpoint, for example, and used to determine the very recent history of the function. This option can be used with or without printed trace output. Each function can be given its own *pdl*, or one *pdl* can serve several functions.

**:entryprint** *form*

*form* is evaluated and the value is included in the trace message for calls to the function. You can give this option more than once, and all the values will appear, preceded by **\\**.

**:exitprint** *form*

*form* is evaluated and the value is included in the trace message for returns from the function. You can give this option more than once, and all the values will appear, preceded by **\\**.

**:print** *form*

*form* is evaluated and the value is included in the trace messages for both calls to and returns from the function. You can give this option more than once, and all the values will appear, preceded by **\\**.

**:entry** *list*

Specifies a list of arbitrary forms whose values are printed along with the usual entry-trace. The list of resultant values, when printed, is preceded by **\\** to separate it from the other information.

**:exit** *list*

Similar to **:entry**, but specifies expressions whose values are printed with the exit-trace. The list of values printed is preceded by **\\**.

**:arg :value :both nil**

Specifies which of the usual trace printouts should be enabled.

<i>If you specify</i>	<i>Then</i>
<b>:arg</b>	On function entry prints the name of the function and the values of its arguments.
<b>:value</b>	On function exit prints the returned value(s) of the function.
<b>:both</b>	Same as if both <b>:value</b> and <b>:arg</b> were specified.
<b>nil</b>	Same as if neither <b>:value</b> or <b>:arg</b> was specified.
None	The default is to <b>:both</b> .

If any further *options* appear after one of these, they are not treated as options. Rather, they are considered to be arbitrary forms whose values are to be printed on entry and/or exit to the function, along with the normal trace information. The values printed are preceded by a //, and follow any values specified by **:entry** or **:exit**. Note that since these options "swallow" all following options, if one is given it should be the last option specified.

If the variable **arglist** is used in any of the expressions given for the **:cond**, **:break**, **:entry**, or **:exit** options, or after the **:arg**, **:value**, **:both**, or **nil** option, when those expressions are evaluated the value of **arglist** will be bound to a list of the arguments given to the traced function. Thus the following form would cause a break in **foo** if and only if the first argument to **foo** is **nil**.

```
(trace (foo :break (null (car arglist))))
```

If the **:break** or **:error** option is used, the variable **arglist** will be valid inside the break-loop. If you **setq arglist**, the arguments seen by the function will change.

Similarly, the variable **values** will be a list of the resulting values of the traced function. For obvious reasons, this should only be used with the **:exit** option. If the **:exitbreak** option is used, the variables **values** and **arglist** are valid inside the break-loop. If you **setq values**, the values returned by the function will change.

You can "factor" the trace specifications, as explained earlier. For example,

```
(trace ((foo bar) :break (bad-p arglist) :value))
```

is equivalent to

```
(trace (foo :break (bad-p arglist) :value)
      (bar :break (bad-p arglist) :value))
```

Since a list as a function name is interpreted as a list of functions, nonatomic function names are specified as follows:

```
(trace (:function (:method flavor :message) :break t))
```

(See the section "Function Specs".)

**trace-compile-flag***Variable*

If the value of **trace-compile-flag** is **non-nil**, the functions created by **trace** will get compiled, allowing you to trace special forms such as **cond** without interfering with the execution of the tracing functions. The default value of this flag is **nil**.

**6.2 Controlling the Format of trace Output**

Tracing output is printed on the stream that is the value of **trace-output**. This is synonymous with **terminal-io** unless you change it. Following is an example of the default form of **trace** output:

```

1 Enter FACT 4.
| 2 Enter FACT 3.
|   3 Enter FACT 2.
|   | 4 Enter FACT 1.
|   |   5 Enter FACT 0.
|   |   5 Exit FACT 1.
|   | 4 Exit FACT 1.
|   3 Exit FACT 2.
| 2 Exit FACT 6.
1 Exit FACT 24.

```

You can use the variables **si:\*trace-columns-per-level\***, **si:\*trace-bar-p\***, **si:\*trace-bar-rate\***, and **si:\*trace-old-style\*** to control the format of **trace** output.

**si:\*trace-columns-per-level\****Variable*

For **trace** output, controls the number of columns of indentation that are added for each level of function call. The value must be an integer. The default is 2.

**si:\*trace-bar-p\****Variable*

For **trace** output, controls whether columns of vertical bars are printed. If the value is not **nil**, they are printed; otherwise, spaces are printed instead of the vertical bars. The default is **t** (print the bars).

**si:\*trace-bar-rate\****Variable*

When **si:\*trace-bar-p\*** is not **nil**, columns of vertical bars are printed in **trace** output for every *n* levels of function call, where *n* is the value. The value must be an integer. The default is 2.

**si:\*trace-old-style\****Variable*

If not **nil**, the old, Maclisp-compatible form of printing **trace** output is used. The default is **nil** (use the new style).

### 6.3 Untracing Function Execution

**untrace** &quote &rest *fns*

*Special Form*

Use **untrace** to undo the effects of **trace** and restore functions *fns* to their normal, untraced state. **untrace** takes multiple specifications, for example, (**untrace foo bar baz**). Calling **untrace** with no arguments untraces all functions currently being traced.

## 7. Advising a Function

To *advise* a function is to tell a function to do something extra in addition to its actual definition. Advising is achieved by means of the function **advise**. The something extra is called a piece of advice, and it can be done before, after, or around the definition itself. The advice and the definition are independent, in that changing either one does not interfere with the other. Each function can be given any number of pieces of advice.

Advising is fairly similar to tracing, but its purpose is different. Tracing is intended for temporary changes to a function to give the user information about when and how the function is called and when and with what value it returns. Advising is intended for semipermanent changes to what a function actually does. The differences between tracing and advising are motivated by this difference in goals.

Advice can be used for testing out a change to a function in a way that is easy to retract. In this case, you would call **advise** from the terminal. It can also be used for customizing a function that is part of a program written by someone else. In this case you would be likely to put a call to **advise** in one of your source files or your login init file rather than modifying the other person's source code. See the section "Logging in".

Advising is implemented with encapsulation, so if the function is redefined (for example, with **defun** or by loading it from a compiled code file), the advice will be transferred from the old definition to the new definition. See the section "Encapsulations".

**advise** *function class name position &body forms* *Special Form*

A function is advised by the special form

(*advise function class name position  
form1 form2...*)

None of this is evaluated.

*function* Specifies the function to put the advice on. It is usually a symbol, but any function spec is allowed. (See the section "Function Specs".)

*class* Specifies either **:before**, **:after**, or **:around**, and says when to execute the advice (before, after, or around the execution of the definition of the function). The meaning of **:around** advice is explained a couple of sections below.

*name* Specifies an arbitrary symbol that is remembered as the name of this particular piece of advice. It is used to keep track of multiple pieces of advice on the same function. If you have no name in mind, use **nil**; then we say the piece of advice is anonymous.

A given function and class can have any number of pieces of anonymous advice, but it can have only one piece of named advice for any one name. If you try to define a second one, it replaces the first.

Advice for testing purposes is usually anonymous. Advice used for customizing someone else's program should usually be named so that multiple customizations to one function have separate names. Then, if you reload a customization that is already loaded, it does not get put on twice.

*position* Specifies where to put this piece of advice in relation to others of the same class already present on the same function.

Position can have these values:

- *position* can be **nil**. The new advice goes in the default position: it usually goes at the beginning (where it is executed before the other advice), but if it is replacing another piece of advice with the same name, it goes in the same place that the old piece of advice was in.
- *position* can be a number, which is the number of pieces of advice of the same class to precede this one. For example, 0 means at the beginning; a very large number means at the end.
- *position* can have the name of an existing piece of advice of the same class on the same function; the new advice is inserted before that one.

*forms* Specifies the advice; they get evaluated when the function is called.

Example: The following form modifies the factorial function so that if it is called with a negative argument it signals an error instead of running forever.

```
(advise factorial :before negative-arg-check nil
 (if (minusp (first arglist))
  (ferror "factorial of negative argument")))
```

**unadvise** &optional *function class position*

*Special Form*

Removes pieces of advice. None of its subforms are evaluated. *function* and *class* have the same meaning as they do in the function **advise**. *position* specifies which piece of advice to remove. It can be the numeric index (0 means the first one) or it can be the name of the piece of advice.

**unadvise** can remove more than one piece of advice if some of its arguments are missing or **nil**. The arguments *function*, *class*, and *position* all act independently. A missing value or **nil** means all possibilities for that aspect of advice. For example, the following form removes all **:before**, **:after**, and **:around** advice named **negative-arg-check** on the **factorial** function.

(unadvise factorial nil negative-arg-check)

In this example **unadvise** removes all **:around** advice on all functions in all positions with all names.

(unadvise nil :around)

In this example **unadvise** removes all classes of advice named **my-personal-advice** on all functions.

(unadvise nil nil my-personal-advice)

(**unadvise**) removes all advice on all functions, since *function*, *class*, and *position* take on all possible values.

The following are the primitive functions for adding and removing advice. Unlike the special forms **advise** and **unadvise**, the following are functions and can be conveniently used by programs. **advise** and **unadvise** are actually macros that expand into calls to these two.

**si:advise-1** *function class name position forms* *Function*

Adds advice. The arguments have the same meaning as in **advise**. Note that the *forms* argument is *not* a **&rest** argument.

**si:unadvise-1** *function &optional class position* *Function*

Removes advice. *function*, *class*, and *position* are independent. If *function*, *class*, or *position* is **nil**, or if *class* or *position* is unspecified, all classes of advice or advice for all functions, at all positions, or with all names is removed.

You can find out manually what advice a function has with **grindef**, which grinds the advice on the function as forms that are calls to **advise**. These are in addition to the definition of the function.

To poke around in the advice structure with a program, you must work with the encapsulation mechanism's primitives. See the section "Encapsulations".

**si:advised-functions**

*Variable*

A list of all functions that have been advised.

## 7.1 Designing the Advice

For advice to interact usefully with the definition and intended purpose of the function, it must be able to interface to the data flow and control flow through the function. The system provides conventions for doing this.

The list of the arguments to the function can be found in the variable **arglist**. **:before** advice can replace this list, or an element of it, to change the arguments passed to the definition itself. If you replace an element, it is wise to copy the whole list first with:



```
(setq arglist (copylist arglist))
```

After the function's definition has been executed, the list of the values it returned can be found in the variable **values**. **:after** advice can set this variable or replace its elements to cause different values to be returned.

All the advice is executed within a **prog**, so any piece of advice can exit the entire function and return some values with **return**. No further advice will be executed. If a piece of **:before** advice does this, then the function's definition will not even be called.

## 7.2 :around Advice

A piece of **:before** or **:after** advice is executed entirely before or entirely after the definition of the function. **:around** advice is wrapped around the definition; that is, the call to the original definition of the function is done at a specified place inside the piece of **:around** advice. You specify where by putting the symbol **:do-it** in that place.

For example, **(+ 5 :do-it)** as a piece of **:around** advice would add 5 to the value returned by the function. This could also be done by the following:

```
(setq values (list (+ 5 (car values))))
```

as **:after** advice.

When there is more than one piece of **:around** advice, they are stored in a sequence just like **:before** and **:after** advice. Then, the first piece of advice in the sequence is the one started first. The second piece is substituted for **:do-it** in the first one. The third one is substituted for **:do-it** in the second one. The original definition is substituted for **:do-it** in the last piece of advice.

**:around** advice can access **arglist**, but **values** is not set up until the outermost **:around** advice returns. At that time, it is set to the value returned by the **:around** advice. It is reasonable for the advice to receive the values of the **:do-it** (for example, with **multiple-value-list**) and play with them before returning them (for example, with **values-list**).

**:around** advice can **return** from the **prog** at any time, whether the original definition has been executed yet or not. It can also override the original definition by failing to contain **:do-it**. Containing two instances of **:do-it** can be useful under peculiar circumstances. If you are careless, however, the original definition might be called twice, but something like the following certainly works reasonably.

```
(if (foo) (+ 5 :do-it) (* 2 :do-it))
```



within another is removed. Other pieces of advice, which have been placed on one function and not limited to within another, are not removed.

**(unadvise)** removes absolutely all advice, including advice for one function within another.

The function versions of **advise-within** and **unadvise-within** are called **si:advise-within-1** and **si:unadvise-within-1** respectively. **advise-within** and **unadvise-within** are macros that expand into calls to the other two.

## 8. Stepping Through an Evaluation

The step facility gives you the ability to follow every step of the evaluation of a form and examine what is going on. It is analogous to a single-step proceed facility often found in machine-language debuggers. Use the step facility if your program is behaving strangely, and it is not obvious how it is getting into this strange state. See the section "Stepping".

You can enter the stepper in two ways:

- Use the **step** function.
- Use the **:step** option of **trace**.

### **step** *form*

*Function*

**step** evaluates *form* with single stepping. It returns the value of *form*.

For example, if you have a function named **foo**, and typical arguments to it might be **t** and **3**, you could say

```
(step '(foo t 3))
```

If a function is traced with the **:step** option, then whenever that function is called it will be single stepped. See the section "Options to **trace**". Note that any function to be stepped must be interpreted; that is, it must be a lambda-expression. Compiled code cannot be handled by the stepper.

When evaluation is proceeding with single stepping, before any form is evaluated, it is (partially) printed out, preceded by a right-facing arrow ( $\rightarrow$ ) character. When a macro is expanded, the expansion is printed out preceded by a double arrow ( $\Rightarrow$ ) character. When a form returns a value, the form and the values are printed out preceded by a left-facing arrow ( $\leftarrow$ ) character; if more than one value is being returned, an and-sign ( $\wedge$ ) character is printed between the values.

Since the forms can be very long, the stepper does not print all of a form; it truncates the printed representation after a certain number of characters. Also, to show the recursion pattern of who calls whom in a graphic fashion, it indents each form proportionally to its level of recursion.

After the stepper prints any of these things, it waits for a command from you. A variety of commands exist to tell the stepper how to proceed, or to look at what is happening.

- |                   |  |
|-------------------|--|
| <b>c-N</b> (Next) | Steps to the next thing. The stepper continues until the next thing to print out, and it accepts another command.                            |
| <b>SPACE</b>      | Goes to the next thing at this level. In other words, it continues to evaluate at this level, but does not step anything at lower levels. In |

- this way you can skip over parts of the evaluation that do not interest you.
- c-U (Up)** Continues evaluating until we go up one level. Similar to the **SPACE** command; it skips over anything on the current level as well as lower levels.
- c-X (eXit)** Exits; finishes evaluating without any more stepping.
- c-T (Type)** Retypes the current form in full (without truncation).
- c-G (Grind)** Grinds (that is, pretty-prints) the current form.
- c-E (Editor)** Enters the editor.
- c-B (Breakpoint)** This command puts you into a breakpoint (that is, a read-eval-print loop) from which you can examine the values of variables and other aspects of the current environment. From within this loop, the following variables are available:
- step-form** The current form.
  - step-values** The list of returned values.
  - step-value** The first returned value.
- If you change the values of these variables, it will work.
- c-L** Clears the screen and redisplay the last ten pending forms (forms being evaluated).
- m-L** Like **c-L**, but does not clear the screen.
- c-m-L** Like **c-L**, but redisplay all pending forms.
- ? or HELP** Prints documentation on these commands.

It is strongly suggested that you write a little function and try the stepper on it. If you get a feel for what the stepper does and how it works, you will be able to tell when it is the right thing to use to find bugs.

## 9. evalhook

The **evalhook** facility provides a "hook" into the evaluator; it is a way you can get a Lisp form of your choice to be executed whenever the evaluator is called. The stepper uses **evalhook**; however, if you want to write your own stepper or something similar, then use this primitive albeit complex facility to do so.

### **evalhook**

*Variable*

If the value of **evalhook** is non-**nil**, then special things happen in the evaluator. When a form (any form, even a number or a symbol) is to be evaluated, **evalhook** is bound to **nil** and the function that was **evalhook**'s value is applied to one argument — the form that was trying to be evaluated. The value it returns is then returned from the evaluator.

**evalhook** is bound to **nil** by **break** and by the Debugger, and **setq**ed to **nil** when errors are dismissed by throwing to the Lisp top-level loop. This provides the ability to escape from this mode if something bad happens.

In order not to impair the efficiency of the Lisp interpreter, several restrictions are imposed on **evalhook**. It only applies to evaluation — whether in a read-eval-print loop, internally in evaluating arguments in forms, or by explicit use of the function **eval**. It does *not* have any effect on compiled function references, on use of the function **apply**, or on the "mapping" functions. (In Zetalisp, as opposed to Maclisp, it is not necessary to do **(\*rset t)** nor **(sstatus evalhook t)**. Also, Maclisp's special-case check for **store** is not implemented.)

### **evalhook form evalhook &optional applyhook**

*Function*

**evalhook** is a function that helps exploit the **evalhook** feature. The *form* is evaluated with **evalhook** lambda-bound to the function *evalhook*. The checking of **evalhook** is bypassed in the evaluation of *form* itself, but not in any subsidiary evaluations, for instance of arguments in the *form*. This is like a "one-instruction proceed" in a machine-language debugger.

```

Example:
;; This function evaluates a form while printing debugging
;; information.
(defun hook (x)
  (terpri)
  (evalhook x 'hook-function))

;; Notice how this function calls evalhook to evaluate the
;; form f, so as to hook the subforms.
(defun hook-function (f)
  (let ((v (evalhook f 'hook-function)))
    (format t "form: ~s~Xvalue: ~s~X" f v)
    v))

;; This isn't a very good program, since if f returns multiple
;; values, it will not work.

```

The following output might be seen from (**hook '(cons (car '(a . b)) 'c)**):

```

form: (quote (a . b))
value: (a . b)
form: (car (quote (a . b)))
value: a
form: (quote c)
value: c
(a . c)

```

Normally after **eval** has evaluated the arguments to a function, it calls the function. If **applyhook** exists, however, **eval** calls the hook with two arguments: the function and its list of arguments. The values returned by the hook constitute the values for the form. The hook could use **apply** on its arguments to do what **eval** would have done normally. This hook is active for special forms as well as for real functions.

Whenever either an **evalhook** or **applyhook** is called, both hooks are bound off. The **evalhook** itself can be **nil** if only an **applyhook** is needed.

**applyhook** catches only **apply** operations done by **eval**. It does not catch **apply** called in other parts of the interpreter or **apply** or **funcall** operations done by other functions such as **mapcar**. In general, such uses of **apply** can be dealt with by intercepting the call to **mapcar**, using the **applyhook**, and substituting a different first argument.

The argument list is like an **&rest** argument: it might be stack-allocated but is not guaranteed to be. Hence you cannot perform side-effects on it and you cannot store it in any place that does not have the same dynamic extent as the call to **applyhook**.

## 9.1 applyhook

**applyhook** provides a hook into **apply**, much as **evalhook** provides a hook into **eval**.

### **applyhook**

*Variable*

When the value of this variable is not **nil** and **eval** calls **apply**, **applyhook** is bound to **nil** and the function that was its value is applied to two arguments: the function that **eval** gave to **apply** and the list of arguments to that function. The value it returns is returned from the evaluator.

### **applyhook** *function args evalhook applyhook*

*Function*

*function* is applied to *args* with **evalhook** lambda-bound to the function *evalhook* and with **applyhook** lambda-bound to the function *applyhook*. Like the **evalhook** function, this bypasses the first place where the relevant hook would normally be triggered. Either of the last two arguments can be **nil**.





## 10. The MAR

The MAR facility exists only on the LM-2. The 3600 has no identical or equivalent facility.

The MAR facility allows any word or contiguous set of words to be monitored constantly, and can cause an error if the words are referenced in a specified manner. The name MAR derives from a similar device on the ITS PDP-10s and is an acronym for Memory Address Register. The MAR checking is done by the Lisp Machine's memory management hardware, and so the speed of general execution when the MAR is enabled is not significantly slowed down. However, the speed of accessing pages of memory containing the locations being checked is slowed down somewhat, since every reference involves a microcode trap.

The MAR is controlled by the following functions:

**dbg:set-mar** *location cycle-type &optional n-words '1* *Function*

The **dbg:set-mar** function clears any previous setting of the MAR, and sets the MAR on *n-words* words, starting at *location*. *location* can be any object. Often it will be a locative pointer to a cell, probably created with the **locf** special form. *n-words* currently defaults to 1. *cycle-type* determines under what conditions to trap and can have the following values:

<b>:read</b>	Only reading the location should cause an error.
<b>:write</b>	Only writing the location should cause an error.
<b>t</b>	Both reading and writing the location should cause an error.

To set the MAR to detect **setq** (and binding) of the variable **foo**, use:

```
(dbg:set-mar (value-cell-location 'foo) ':write)
```

**dbg:clear-mar** *Function*

Turns off the MAR. Warm booting the machine disables the MAR but does not turn it off; that is, references to the MARed pages are still slowed down. **dbg:clear-mar** does not speed things back up until the next time the pages are swapped out.

**dbg:mar-mode** *Function*

(**dbg:mar-mode**) returns a symbol indicating the current state of the MAR. It returns one of the following:

<b>nil</b>	The MAR is not set.
<b>:read</b>	The MAR causes an error if there is a read.

**:write** The MAR causes an error if there is a write.

**t** The MAR causes an error if there is any reference.

Note that using the MAR makes the pages on which it is set somewhat slower to access, until the next time they are swapped out and back in again after the MAR is shut off. Also, use of the MAR currently breaks the read-only feature if those pages were read-only.

Proceeding from a MAR break allows the memory reference that got an error to take place, and continues the program with the MAR still effective. When proceeding from a write, the Debugger asks you whether to allow the write to take place or to inhibit it, leaving the location with its old contents.

Most — but not all — write operations first do a read. **setq** and **rplaca** are examples. This means that if the MAR is in **:read** mode it will catch writes as well as reads; however, they will trap during the reading phase, and consequently the data to be written will not be displayed. This also means that setting the MAR to **t** mode causes most writes to trap twice, first for a read and then again for a write. So when the MAR says that it trapped because of a read, this means a read at the hardware level, which might not look like a read in your program.

## 11. Variable Monitoring

Variable monitoring works only on the LM-2.

**monitor-variable** *sym* &optional *current-value-cell-only-p* *monitor-function* *Function*

Calls a given function just after *sym* is **setq**ed (by compiled code or otherwise). Does not trigger on binding of *sym*. The function is given both the old and new values as arguments. It does not get *sym*, the name of then variable, as an argument, so it is usually necessary to use a closure as *monitor-function* in order to remember this. The old value is **nil** if *sym* had been unbound.

The default monitoring function prints *sym* and the old and new values. This behavior can be changed by specifying the *monitor-function* argument.

Normally this feature applies to all **setqs**, but if *current-value-cell-only-p* is specified non-**nil**, it applies only to those **setqs** that would alter *sym*'s currently active value cell. This is only relevant when *sym* is subject to a closure.

Do not try to use this feature with variables that are forwarded to A-memory (for example, **inhibit-scheduling-flag**).

**unmonitor-variable** &optional *sym* *Function*

If *sym* is being monitored, it is restored to normal. If no *sym* is specified, all variables that have been monitored are unmonitored.



## Index

### A

**A**  
 :around  
 Designing the  
 si:  
 si:  
 Summary of Debugging  
 Advising One Function Within  
 dbg:  
 dbg:  
 Hook

### A

ABORT Debugger command 9, 15  
 Advice 28  
 Advice 27  
 Advice to functions 25  
 advise special form 25  
 advise-1 function 27  
 advise-within special form 29  
 advised-functions variable 27  
 Advising a Function 25  
 Advising One Function Within Another 29  
 Aids 17  
 Another 29  
 applyhook 35  
 applyhook function 35  
 applyhook variable 35  
 arg 13  
 arg function 13  
 :arg option to trace 20  
 arglist variable 20  
 :argpdl option to trace 20  
 arguments 34  
 :around Advice 28

### A

### B

### B

Backtrace 9, 15  
 Backtrace information 14  
 Backtrace of the call stack 14  
 :both option to trace 20  
 :break option to trace 20  
 bug function 9  
 Bug mail 9, 15  
 Bug reports 9, 15

### B

### C

### C

c-A Debugger command 15  
 c-B Debugger command 9, 15  
 c-C Debugger command 9, 15  
 c-E Debugger command 9, 15  
 c-G Debugger command 5  
 c-HELP Debugger command 9, 15  
 c-L Debugger command 9, 15  
 c-M Debugger command 9, 15  
 c-m-A Debugger command 9, 15  
 c-m-B Debugger command 9, 15  
 c-m-F Debugger command 9, 15  
 c-m-H Debugger command 9, 15  
 c-m-L Debugger command 9, 15  
 c-m-N Debugger command 9, 15  
 c-m-P Debugger command 9, 15

### C







c-T	Debugger command	9, 15
c-X	Debugger command	9, 15
HELP	Debugger command	9, 15
m-<	Debugger command	9, 15
m->	Debugger command	9, 15
m-B	Debugger command	9, 15
m-C	Debugger command	9, 15
m-I	Debugger command	9, 13, 15
m-L	Debugger command	9, 15
m-N	Debugger command	9, 15
m-P	Debugger command	9, 15
m-S	Debugger command	5, 9, 15
m-X	Debugger command	9, 15
RESUME	Debugger command	9, 15
	Debugger Commands	9, 15
Summary of	Debugger Commands	15
	Debugger functions to return values in current stack	
	frame	13
Summary of	Debugging Aids	17
<b>dbg:</b>	<b>*defer-package-dwim*</b> variable	14
	Designing the Advice	27
Error	Display	2

## E

## E

## E

	Entering the Debugger	1
	<b>:entry</b> option to <b>trace</b>	20
	<b>:entrycond</b> option to <b>trace</b>	20
	<b>:entryprint</b> option to <b>trace</b>	20
	Error Display	2
	<b>:error</b> option to <b>trace</b>	20
	<b>evalhook</b>	33
	<b>evalhook</b> function	33, 34
	<b>evalhook</b> variable	33
Stepping Through an	Evaluation	31
	Examining values of instance variables	13
Tracing Function	Execution	19
Untracing Function	Execution	24
	<b>:exit</b> option to <b>trace</b>	20
	<b>:exitbreak</b> option to <b>trace</b>	20
	<b>:exitcond</b> option to <b>trace</b>	20
	<b>:exitprint</b> option to <b>trace</b>	20

## F

## F

## F

	<b>advise</b> special	form	25
	<b>advise-within</b> special	form	29
	<b>trace</b> special	form	19
	<b>unadvise</b> special	form	26
	<b>unadvise-within</b> special	form	29
	<b>untrace</b> special	form	24
	Controlling the	Format of <b>trace</b> Output	23
	Current stack	frame	9, 15
Debugger functions to return values in current stack		frame	13
	<b>dbg:</b>	<b>*frame*</b> variable	13
	<b>dbg:</b>	<b>fun</b>	13
	<b>dbg:</b>	<b>fun</b> function	13
Advising a		Function	25

<b>applyhook</b>	function	35
<b>bug</b>	function	9
<b>dbg</b>	function	1
<b>dbg:arg</b>	function	13
<b>dbg:clear-mar</b>	function	37
<b>dbg:fun</b>	function	13
<b>dbg:loc</b>	function	13
<b>dbg:mar-mode</b>	function	37
<b>dbg:set-mar</b>	function	37
<b>dbg:val</b>	function	13
<b>evalhook</b>	function	33, 34
<b>monitor-variable</b>	function	39
<b>si:advise-1</b>	function	27
<b>si:unadvise-1</b>	function	27
<b>step</b>	function	31
<b>unmonitor-variable</b>	function	39
Tracing	Function Execution	19
Untracing	Function Execution	24
Advising One	Function Within Another	29
Advice to Debugger	functions to return values in current stack frame	25
	Functions used inside the Debugger	13

**H**

**H**

**H**

HELP Debugger command	9, 15
Hook arguments	34
How to Use the Debugger	5

**I**

**I**

**I**

Backtrace	information	14
Functions used	inside the Debugger	13
Examining values of	instance variables	13

**L**

**L**

**L**

<b>dbg: loc</b>	13
<b>dbg: loc</b>	function 13

**M**

**M**

**M**

<b>m-&lt;</b>	Debugger command	9, 15
<b>m-&gt;</b>	Debugger command	9, 15
<b>m-B</b>	Debugger command	9, 15
<b>m-C</b>	Debugger command	9, 15
<b>m-I</b>	Debugger command	9, 13, 15
<b>m-L</b>	Debugger command	9, 15
<b>m-N</b>	Debugger command	9, 15
<b>m-P</b>	Debugger command	9, 15
<b>m-S</b>	Debugger command	5, 9, 15
<b>m-X</b>	Debugger command	9, 15
Bug	mail	9, 15
	Manipulating the control stack	9, 15
The	MAR	37
<b>dbg: mar-mode</b>	function	37
<b>dbg: monitor-variable</b>	function	39

Variable Monitoring 39  
Monitoring the value of a variable 27

O

O

O

Advising One Function Within Another 29  
 :arg option to trace 20  
 :argpdl option to trace 20  
 :both option to trace 20  
 :break option to trace 20  
 :cond option to trace 20  
 :entry option to trace 20  
 :entrycond option to trace 20  
 :entryprint option to trace 20  
 :error option to trace 20  
 :exit option to trace 20  
 :exitbreak option to trace 20  
 :exitcond option to trace 20  
 :exitprint option to trace 20  
 :print option to trace 20  
 :step option to trace 20  
 :value option to trace 20  
 :wherein option to trace 20  
 Controlling the Format of trace  
 trace Options to trace 20  
 Output 23  
 output 23

P

P

P

:print option to trace 20

R

R

R

Bug reports 9, 15  
 RESUME Debugger command 9, 15  
 Debugger functions to return values in current stack frame 13

S

S

S

dbg: set-mar function 37  
 dbg: \*show-backtrace\* variable 14  
 si:\*trace-bar-p\* variable 23  
 si:\*trace-bar-rate\* variable 23  
 si:\*trace-columns-per-level\* variable 23  
 si:\*trace-old-style\* variable 23  
 si:advise-1 function 27  
 si:advised-functions variable 27  
 si:unadvise-1 function 27  
 advise special form 25  
 advise-within special form 29  
 trace special form 19  
 unadvise special form 26  
 unadvise-within special form 29  
 untrace special form 24  
 Backtrace of the call stack 14  
 Manipulating the control stack 9, 15  
 Current stack frame 9, 15

Debugger functions to return values in current  
 stack frame 13  
**step** function 31  
**:step** option to **trace** 20  
 Stepping Through an Evaluation 31  
 Summary of Debugger Commands 15  
 Summary of Debugging Aids 17

T

T

T

**:arg** option to **trace** 20  
**:argpdi** option to **trace** 20  
**:both** option to **trace** 20  
**:break** option to **trace** 20  
**:cond** option to **trace** 20  
**:entry** option to **trace** 20  
**:entrycond** option to **trace** 20  
**:entryprint** option to **trace** 20  
**:error** option to **trace** 20  
**:exit** option to **trace** 20  
**:exitbreak** option to **trace** 20  
**:exitcond** option to **trace** 20  
**:exitprint** option to **trace** 20  
**:print** option to **trace** 20  
**:step** option to **trace** 20  
**:value** option to **trace** 20  
**:wherein** option to **trace** 20  
 Options to **trace** 20  
**trace** output 23  
 Controlling the Format of **trace** Output 23  
**trace** special form 19  
**sl:** **\*trace-bar-p\*** variable 23  
**sl:** **\*trace-bar-rate\*** variable 23  
**sl:** **\*trace-columns-per-level\*** variable 23  
**trace-compile-flag** variable 23  
**sl:** **\*trace-old-style\*** variable 23  
 Tracing Function Execution 19

U

U

U

**unadvise** special form 26  
**sl:** **unadvise-1** function 27  
**unadvise-within** special form 29  
**unmonitor-variable** function 39  
**untrace** special form 24  
 Untracing Function Execution 24  
 Functions used inside the Debugger 13

V

V

V

**dbg:** **val** 13  
**dbg:** **val** function 13  
 Monitoring the value of a variable 27  
 Debugger functions to return **:value** option to **trace** 20  
 Examining values in current stack frame 13  
 values of instance variables 13  
**values** variable 20  
**applyhook** variable 35

<b>arglist</b>	variable	20
<b>dbg:*debug-io-override*</b>	variable	14
<b>dbg:*defer-package-dwim*</b>	variable	14
<b>dbg:*frame*</b>	variable	13
<b>dbg:*show-backtrace*</b>	variable	14
<b>evalhook</b>	variable	33
Monitoring the value of a	variable	27
<b>si:*trace-bar-p*</b>	variable	23
<b>si:*trace-bar-rate*</b>	variable	23
<b>si:*trace-columns-per-level*</b>	variable	23
<b>si:*trace-old-style*</b>	variable	23
<b>si:advised-functions</b>	variable	27
<b>trace-compile-flag</b>	variable	23
<b>values</b>	variable	20
	Variable Monitoring	39
Examining values of instance	variables	13

**W****W****W**

	<b>:wherein</b> option to <b>trace</b>	20
Advising One Function	Within Another	29

*symbolics*™

---

# **MAINT** Maintaining Large Systems

---

---

Cambridge, Massachusetts

---

# **Maintaining Large Systems**

**# 995005**

**February 1984**

**This document corresponds to Release 5.0.**

This document was prepared by the Documentation Group of Symbolics, Inc.

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Symbolics is a trademark of Symbolics, Inc., Cambridge, Massachusetts.

Copyright © 1981, 1979, 1978 Massachusetts Institute of Technology.  
All rights reserved.

Enhancements copyright © 1984, 1983, 1982 Symbolics, Inc. of Cambridge,  
Massachusetts.

All rights reserved. Printed in USA.

This document may not be reproduced in whole or in part without the prior written consent of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

## **Table of Contents**

	<b>Page</b>
<b>1. Introduction to Making a System</b>	<b>1</b>
<b>2. Defining a System</b>	<b>3</b>
<b>3. Transformations</b>	<b>7</b>
<b>4. Loading the System Definition</b>	<b>11</b>
<b>5. Making a System</b>	<b>13</b>
5.1 <b>make-system</b> keywords	<b>13</b>
5.2 Using the <b>:version</b> and <b>:update-directory</b> keywords	<b>17</b>
5.3 Maintaining Parallel Systems for the LM-2 and the 3600	<b>19</b>
5.4 Describing a System	<b>19</b>
<b>6. Adding New Keywords to make-system</b>	<b>21</b>
<b>7. Adding New Options to defsystem</b>	<b>23</b>
<b>8. More Esoteric Transformations</b>	<b>25</b>
<b>9. Patch Facility</b>	<b>27</b>
9.1 Finding Out About Patchable Systems	<b>29</b>
9.2 Types of Patch Files	<b>30</b>
9.3 Loading Patches	<b>33</b>
9.4 Making Patches	<b>34</b>
9.5 Patchable System Status	<b>37</b>
<b>Index</b>	<b>39</b>



## 1. Introduction to Making a System

When a program gets large, it is often desirable to split it up into several files. One reason is to help keep the parts of the program organized, to make things easier to find. Another is that programs broken into small pieces are more convenient to edit and compile. It is particularly important to avoid the need to recompile all of a large program every time any piece of it changes; if the program is broken up into many files, only the files that have changes in them need to be recompiled.

The apparent drawback to splitting up a program is that more mechanism is needed to manipulate it. To load the program, you now have to load several files separately, instead of just loading one file. To compile it, you have to figure out which files need compilation, by seeing which have been edited since they were last compiled, and then you have to compile those files.

An even more complicated factor is that files can have interdependencies. You might have a file called "defs" that contains some macro definitions (or flavor or structure definitions), and functions in other files might use those macros. This means that in order to compile any of those other files, you must first load the file "defs" into the Lisp environment, so that the macros will be defined and can be expanded at compile time. You would have to remember this whenever you compile any of those files. Furthermore, if "defs" has changed, other files of the program might need to be recompiled because the macros might have changed and need to be reexpanded.

This chapter describes the *system* facility, which takes care of all these conditions for you. The way it works is that you define a set of files to be a *system*, using the **defsystem** special form. See the section "Defining a System". This system definition includes the following:

- Which files make up the system.
- Which files depend on the presence of others.
- What properties the system should have, for example, the package into which the object code should be compiled, or whether the system can be patched.

You put this system definition into its own little file, and then all you have to do is load that file (or have your init file load it) and the Lisp environment will know about your system and what files are in it. See the section "Loading the System Definition". You can then use the **make-system** function to load all the files of the system, recompile all the files that need compiling, and so on. See the section "Making a System".

The system facility is very general and extensible. This chapter explains how to use it and how to extend it. This chapter also explains the *patch* facility, which lets you conveniently update a large program with incremental changes.



## 2. Defining a System

### **defsystem** *name* &*body options*

*Special Form*

Defines a system named *name*. *options* are keywords and fall into three categories: properties of the system, modules, and transformations. (See the section "Transformations".) The simplest system is a set of files and a transformation to be performed on them.

#### Example 1.

```
(defsystem mysys
  (:compile-load ("q:>george>prog1" "q:>george2>prog2")))
```

#### Example 2.

```
(defsystem zmail
  (:name "Zmail")
  (:pathname-default "q:>zmail>")
  (:package zwei)
  (:module defs "defs")
  (:module mult "mult" :package tv)
  (:module main ("top" "comnds" "mail" "user" "window"
                 "filter" mult "cometh"))
  (:compile-load defs)
  (:compile-load main (:fasload defs)))
```

#### Example 3.

```
(defsystem bar
  (:module reader-macros "rdmac")
  (:module other-macros "macros")
  (:module main-program "main")
  (:compile-load reader-macros)
  (:compile-load other-macros (:fasload reader-macros))
  (:compile-load main-program (:fasload reader-macros
                                       other-macros)))
```

Example 1 defines a new system called **mysys**, which consists of two files, both of which are to be compiled and loaded.

Example 2 is somewhat more complicated. The primary difference is that there is a module **defs** that must be loaded before **main** can be compiled.

Example 3 has two levels of dependency. **reader-macros** must be compiled and loaded before **other-macros** can be compiled. Both **reader-macros** and **other-macros** must then be loaded before **main-program** can be compiled.

All the **defsystem** options, except transformations, are listed here.

- :name** Specifies a "pretty" version of the name for the system, for use in printing.
- :short-name** Specifies an abbreviated name used in constructing disk label comments and in patch file names for some file systems.
- :component-systems**  
Specifies the names of other systems used to make up this system. Performing an operation on a system with component systems is equivalent to performing the same operation on all the individual systems. The format is  
(:component-systems *names...*)
- :package** Specifies the package in which transformations are performed. A package specified here overrides the one specified in the attribute list of the file in question.
- :pathname-default**  
Gives a local default within the definition of the system for strings to be parsed into pathnames. Typically this specifies the directory, when all the files of a system are on the same directory.
- :patchable** Allows the system to be patched. (See the section "Patch Facility".) An optional argument specifies the directory to put patch files in. The default is the **:pathname-default** of the system.
- :initial-status** Specifies what the status of the system should be when **make-system** is used to create a new major version. The default is **:experimental**. (See the section "Patchable System Status".)
- :bug-reports** Specifies the name of the system (a string) to which bug mail can be sent. Supply a documentation string describing the purpose of the bug mail. The name of the system appears in the Bug Mail menu (evoked by clicking middle on [Mail] in Zmail) and the documentation string appears in the mouse documentation line. Example: **:bug-reports "Daedalus" "Report problems with the Daedalus system."** sends mail to Bug-Daedalus.
- :not-in-disk-label**  
Makes a patchable system not appear in the disk label comment. This should probably never be specified for a user system. It is used by patchable systems internal to the main Lisp system, to avoid cluttering up the label.
- :maintaining-sites**  
Specifies the list of sites that maintain the system; declares which sites can patch a system and helps to monitor versions in order to ensure that no changes are lost. This option is meaningful only for patchable systems. For example:

```
(defsystem dla-file-system
  ...
  (:maintaining-sites :mit)
  ...)
```

The default for **:maintaining-sites** when it is undeclared is usually the local site. When you attempt to distribute a system with an undeclared maintaining site, you are warned and urged to supply a maintaining site.

When you attempt to patch a system that is not maintained at your site, you see a warning like the following:

```
System DLA-FILE-SYSTEM is not normally maintained at
this site. Patching it here may result in version skews
and make it difficult for your site to receive
subsequent software updates.
Are you sure you really sure you want to patch it? (Yes
or No)
```

### **:module**

Allows assigning a name to a set of files within the system. You can use this name instead of repeating the filenames. The format is:

```
(:module name module-specification options...)
```

*module-specification* can be any of the following:

A string A file name.

A symbol A module name. It stands for all of the files that are in that module of this system.

An *external module component*

A list of the form (*system-name module-names...*), to specify modules in another system. It stands for all of the files that are in all of those modules.

A list of *module components*

A module component is any *module-specification*.

A list of file names

Used in the case where the names of the input and output files of a transformation are not related according to the standard naming conventions, for example, when a compiled code file has a different name or resides in a different directory than the source file. The file names in the list are used from left to right, thus, the first name is the source file. Each file name after the first in the list is defaulted from the previous one in the list.

To avoid syntactic ambiguity, this is allowed as a module component but not as a module specification.

The **:module** clause takes the **:package** option, which overrides any package specified for the whole system for transformations performed on just this module. Sometimes you have a module that needs to use the packages specified by the files' attribute lists rather than the package declared for the system. You can make the files' package specs override the general one by putting **:package nil** in the module's plist (at the end of the **:module** declaration).

The second **defsystem** example lists three modules. The first two each have only one file, and the third one (**main**) is made up both of files and another module. To take examples of the other possibilities:

```
(:module prog (("q:>george>prog" "q:>george2>prog")))  
(:module foo (defs (zmail defs)))
```

The **prog** module consists of one file, but it lives in two directories, **george** and **george2**. If this were a Lisp program, that would mean that the file "q:>george>prog.lisp" would be compiled into "q:>george2>prog.bin". The **foo** module consists of two other modules: the **defs** module in the same system and the **defs** module in the **zmail** system. It is not generally useful to compile files that belong to other systems; thus, this **foo** module would not normally be the subject of a transformation. However, *dependencies* use modules and need to be able to refer to (depend on) modules of other systems. See the section "Transformations".

### 3. Transformations

A *transformation* is an operation, such as compiling or loading, that takes one or more files and does something to them. Transformations are of two types: simple and complex. A *simple transformation* is a single operation on a file, such as compiling it or loading it. A *complex transformation* takes the output from one transformation and performs another transformation on it, for example, loading the results of compilation.

The general format of a simple transformation is:

*(name input dependencies condition)*

*name*            The name of the transformation to be performed on all the files in the module, or all the output files of the other transformation.

*input*            Usually a module specification or another transformation whose output is used. A module specification can have many different formats, including "anonymous" modules recursively including other modules. Read the description of the **:module** keyword: See the section "Defining a System".

*dependencies*    Optional. *dependencies* is a transformation specification, which is either a list:

*(transformation-name module-names...)*

or a list of such lists. *module-names* is either a symbol that is the name of a module in the current system or a list *(system-name module-names...)*.

*condition*        Optional. *condition* is a predicate that specifies when the transformation should take place. Generally it defaults according to the type of the transformation. For a further discussion of conditions: See the section "More Esoteric Transformations".

A dependency declares that all of the indicated transformations must be performed on the indicated modules before the current transformation itself can take place. Thus, in the last line of the following example, the **defs** module must have the **:fasload** transformation performed on it before the **:compile-load** transformation can be performed on **main**.

```
(defsystem zmail
  (:name "Zmail")
  (:pathname-default "q:>zmail>")
  (:package zwei)
  (:module defs "defs")
  (:module mult "mult" :package tv)
  (:module main ("top" "comnds" "mail" "user" "window"
                 "filter" mult "cometh"))
  (:compile-load defs)
  (:compile-load main (:fasload defs)))
```

The dependency has to be a transformation that was explicitly specified as a transformation in the system definition, not just an action that might have been performed by anything. That is, if you have a dependency **(:fasload foo)**, it means that **(:fasload foo)** is a transformation of your system and you depend on that transformation; it does not simply mean that you depend on **foo** being loaded. It is not sufficient if the action is performed as part of a transformation on an anonymous module constructed of other modules, such as in the second example below. It is sufficient if a complex transformation, such as **:compile-load**, expands into the required transformation on the specified module, such as in the third example below.

For example, the following is correct and works properly:

```
(defsystem foo
  (:module foo "foo")
  (:module bar "bar")
  (:compile-load (foo bar)))
```

But the following example does not work because **foo's :fasload** does not occur. The loading of **foo** is performed only implicitly as part of the **:fasload** transformation on the anonymous module **(foo bar)** implicit in the **(:compile-load (foo bar))**.

```
(defsystem foo
  (:module foo "foo")
  (:module bar "bar")
  (:module blort "blort")
  (:compile-load (foo bar))
  (:compile-load blort (:fasload foo)))
```

You must instead write:

```
(defsystem foo
  (:module foo "foo")
  (:module bar "bar")
  (:module blort "blort")
  (:compile-load foo)
  (:compile-load bar)
  (:compile-load blort (:fasload foo)))
```

In the above example, **(:fasload foo)** is part of the expansion of **(:compile-load foo)**; therefore, it can be used as a dependency.



The defined simple transformations are:

- :fasload** Calls the **si:load-binary-file** function to load the indicated files, which must be compiled code files. The *condition* defaults to **si:file-newer-than-installed-p**, which is **t** if a newer version of the file exists on the file computer than was read into the current environment.
- :readfile** Calls the **readfile** function to read the indicated files. Use this for files that are not to be compiled. *condition* defaults to **si:file-newer-than-installed-p**.
- :compile** Calls the **compiler:compile-file** function to compile the indicated files. *condition* defaults to **si:file-newer-than-file-p**, which returns **t** if the source file has been written more recently than the compiled code file.

A special simple transformation is:

- :do-components** (**:do-components dependencies**) inside a system with component systems causes the *dependencies* to be done before anything in the component systems. This is useful when you have a module of macro files used by all of the component systems.

The defined complex transformations are:

- :compile-load** (**:compile-load input compile-dependencies load-dependencies compile-condition load-condition**) is the same as (**:fasload (:compile input compile-dependencies compile-condition load-dependencies load-condition)**). This is the most commonly used transformation. Everything after *input* is optional.
- :compile-load-init** See the section "More Esoteric Transformations".

Each file name in an input specification can in fact be a list of strings for the case where the source file of a program differs from the binary file in more than just the file type. In fact, every file name is treated as if it were an infinite list of file names with the last file name, or, in the case of a single string, the *only* file name, repeated forever at the end. Each simple transformation takes some number of input file name arguments, and some number of output file name arguments. As transformations are performed, these arguments are taken from the front of the file name list. The input arguments are actually removed, and the output arguments are left as input arguments to the next higher transformation.

To make this clearer, consider having the **:compile-load** transformation performed on the **prog** module:

```
(:module prog (("q:>george>prog" "q:>george2>prog")))
```

This means that **prog** is given as the input to the **:compile** transformation and the output from this transformation is given as the input to the **:fasload** transformation. The **:compile** transformation takes one input file name argument — the name of a Lisp source file — and one output file name argument — the name of the compiled code file. The **:fasload** transformation takes one input file name argument — the name of a compiled code file — and no output file name arguments. So, for the first and only file in the **prog** module, the file name argument list looks like ("q:>george>prog" "q:>george2>prog" "q:>george2>prog" ...). The **:compile** transformation is given arguments of "q:>george>prog" and "q:>george2>prog" and the file name argument list, which it outputs as the input to the **:fasload** transformation, is ("q:>george2>prog" "q:>george2>prog" ...). The **:fasload** transformation then is given its one argument of "q:>george2>prog".

Note that dependencies are neither transitive nor inherited. For example, if module **a** depends on macros defined in module **b**, and therefore needs **b** to be loaded in order to compile, and if **b** has a similar dependency on **c**, then **c** will not be loaded during compilation of **a**. Transformations with these dependencies would be written as follows:

```
(:compile-load a (:fasload b))
(:compile-load b (:fasload c))
```

To say that compilation of **a** depends on both **b** and **c**, you would instead write:

```
(:compile-load a (:fasload b c))
(:compile-load b (:fasload c))
```

If, in addition, **a** depended on **c**, but not **b**, during loading (perhaps **a** contains **defvars** whose initial values depend on functions or special variables defined in **c**) you would write the transformations as follows:

```
(:compile-load a (:fasload b c) (:fasload c))
(:compile-load b (:fasload c))
```

## 4. Loading the System Definition

Typically, you place the system definition (the **defsystem** invocation) in a source file. The file must have the canonical file type of **:lisp**.

**si:set-system-source-file** *system-name source-file* *Function*  
**si:set-system-source-file** allows you to specify *source-file*, the source file that contains the definition of the system called *system-name*, before the system is loaded. *source-file* is loaded the first time that you use **make-system** to load and/or compile your system. *system-name* is the symbol that you supply to **make-system**.

Use **si:set-system-source-file** in your init file.

**make-system** offers to compile and load a new version of the file containing the system definition if it has changed.

Note for users of ITS: This feature of **make-system** works only if the file containing the **defsystem** form has a file type of **lisp**, that is, an FN2 of > on ITS. Thus, if you have a file FOO PKG and want to benefit from using this feature of **make-system**, you should rename the file FOOPKG >.

**make-system** has a feature for finding how out to make a system that has not been defined already. When the system it is looking for has not been defined already or been set up with **si:set-system-source-file**, it looks for system definition information in a file with the following name:

sys: site; *system-name* system

That file should contain **si:set-system-source-file**. For more information: See the document *Software Installation Guide*.



## 5. Making a System

**make-system** *name* &rest *keywords*

*Function*

Compiles and/or loads a system defined by **defsystem**. Consider the following system declaration:

```
(defsystem mysys
  (:compile-load ("q:>george>prog1" "q:>george2>prog2")))
```

If "q:>george>prog1" and "q:>george2>prog2" have both been compiled recently, then **make-system** only loads them as necessary:

```
(make-system 'mysys)
```

**make-system** supports a number of *keyword* options. For example, if any of the constituent files of **mysys** also needs to be compiled, then use:

```
(make-system 'mysys :compile)
```

**make-system** lists what transformations it is going to perform on what files, asks the user for confirmation, then performs the transformations. Prior to each transformation a message is printed listing the transformation being performed, the file to which it is being done, and the package.

Load all twenty-six of them? (Y, N, or S)

If you answer S (meaning *selective*), you are asked for confirmation of each individual transformation.

The behavior of **make-system** can be altered by keywords.

If you run **make-system** on a system that is patchable and not already loaded, **make-system** calls **load-patches** after loading the system. **load-patches** is called with the same options as **make-system**; if **make-system** is specified with the **:silent** keyword, **load-patches** is also silent.

### 5.1 make-system keywords

The **make-system** function recognizes the following keywords:

- :batch** Allows a large compilation to be done unattended. It acts like **:noconfirm** with regard to questions, turns off more-processing and **fdefine-warnings**, and saves the compiler warnings in an editor buffer and a file (it asks you for the name). See the variable **inhibit-fdefine-warnings**. *Figure 16*
- :compile** Compiles files also if necessary. The default is to load but not compile. **:compile** always compiles the newest versions of the system's files. *unless there is a <sup>bin</sup> file already (even of an earlier version)*

- :noconfirm** Assumes a yes answer for all questions that you would otherwise be asked.
- :noload** Does not load any files except those required by dependencies. For use in conjunction with the **:compile** option.
- :noop** Is ignored. This is mainly useful for programs that call **make-system**, so that such programs can include forms like:
- ```
(make-system 'mysys (if compile-p ':compile ':noop))
```
- :nowarn** Suppresses questions requiring operator response. Otherwise you must give permission (yes or no) to have straightforward tasks (like reading files) performed.
- :print-only** Displays the transformations that would be performed; does not actually do any compiling or loading.
- :recompile** Compiles all files, regardless of whether or not they need to be compiled. Has the effect of **:compile** and **:reload**. **:recompile** always compiles the newest versions of each constituent file of a system.
- :reload** Bypasses the specified conditions for performing a transformation. Thus files are compiled even if they have not changed and loaded even if they are not newer than the installed version.
- :selective** Asks the user whether or not to perform each transformation that appears to be needed for each file.
- :silent** Avoids printing out each transformation as it is performed.

In addition to the above keywords, you can use the following options for patchable systems.

- :increment-patch** Increment a patchable system's major version without doing any compilations. See the section "Patch Facility".
- :no-increment-patch** When given along with the **:compile** option, disables the automatic incrementing of the major system version that would otherwise take place. See the section "Patch Facility".
- :version** Loads specific versions of a patchable system, as designated in the system version-directory file: See the section "Types of Patch Files". A system version can be expressed as the newest, released, or latest version; a version number; or version name.
- :version** accepts several keyword arguments. Specify the keyword and its arguments as a list. For example, to load version 34 of **mysys**, invoke:

```
(make-system 'mysys '(:version 34.))
```

| <i>Argument</i>  | <i>Meaning</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>:released</b> | <p>Loads the system designated in the system version-directory file as the released version. See the section "Types of Patch Files". When you do not supply the <b>:version</b>, <b>:compile</b>, or <b>:recompile</b> keyword, <b>make-system</b> loads the released system. If there is no released version, then <b>make-system</b> loads the latest version.</p> <p>Example: To load the released version of <b>george</b>, type:</p> <pre>(make-system 'george '(:version :released))</pre> <p>or just:</p> <pre>(make-system 'george)</pre> <p>Note: The system developer designates a particular version of the system as the released version by using the <b>:update-directory</b> keyword to <b>make-system</b>. (See the <b>:update-directory</b> keyword.)</p>                                                                                                                                                      |
| <b>:latest</b>   | <p>Loads the system designated in the system version-directory file as the latest version. See the section "Types of Patch Files". The most recently compiled version of the system is automatically assigned the designation <b>:latest</b>.</p> <p>Example: On Monday the system developer does the following:</p> <pre>(make-system 'alphabet ':recompile)</pre> <p>This invocation compiles the most up-to-date source files in the <b>alphabet</b> system and then loads each newly compiled file. <b>make-system</b> also automatically updates the system version-directory file, marking Monday's version of <b>alphabet</b> as the latest version.</p> <p>On Tuesday the system developer wants to load the version he compiled the day before; hence:</p> <pre>(make-system 'alphabet '(:version :latest))</pre> <p>System developers typically use the <b>:latest</b> keyword to load systems under development.</p> |
| <b>:newest</b>   | <p>Loads the most recently compiled version of each <i>file</i> of a system. The newest version differs from the latest version when individual files in the system have been compiled by hand. Note that you cannot define or load patches for the newest system.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

**Example:** On Tuesday the system developer loads the latest version of the system **alphabet**, which contains files A.lisp.10, A.bin.10, B.lisp.10, B.bin.10, and so on, to Z.lisp.10, Z.bin.10. The developer makes changes to several functions in A.lisp.10, compiles the file to A.bin.11, and saves the source file, A.lisp.11. On Wednesday the developer wants to test the incremental changes to the system, but, to be cautious, doesn't want to destroy the latest system that was compiled and loaded on Monday. To do so, the developer uses the **:newest** keyword to load a system consisting of the most recently compiled versions of each of the system's files: A.bin.11 and the remaining files, B.bin.10 through Z.bin.10.

```
(make-system 'alphabet '(:version :newest))
```

The latest version remains intact; and the newest version is the most experimental version of the system.

*version-number* Loads a particular major version number of the system.

```
(make-system 'george '(:version 23.))
```

Note the decimal point after the version number.

*version-name* Loads the particular version of the system known as *version-name* in the system version-directory file. See the section "Types of Patch Files". The system maintainer must have previously assigned *version-name* by using the **:update-directory** keyword to **make-system**.

**Example:** The system developer plans to demonstrate the **frog** system to a group of prospective customers from Japan. Aside from the regular debugged version, there is a special version that works in Japanese.

After assigning the version name **:japanese** to this particular version of **frog**, the developer can load it, as follows:

```
(make-system 'frog '(:version :japanese))
```

### **:update-directory**

Updates the system version-directory file for the currently loaded version of the system. This keyword works properly *only* for the loaded version of the system. Use **:update-directory** to assign a



*version-name* to a particular system version or to designate a particular system version as the released version.  
**:update-directory** takes a keyword argument; specify **:update-directory** and its argument as a list. When you specify **:update-directory** without an argument, the default entry made to the system version-directory file is **:latest**.

Example 1: The system developer wants to release the latest version of **george**, version #34, for general use. There is currently no released version. The following invocation loads the latest version of **george** and designates it as the released version.

```
(make-system 'george '(:update-directory :released))
```

The developer could also have given this longer but equivalent form:

```
(make-system 'george '(:version 34.) '(:update-directory :released))
```

Example 2: The system developer plans to demonstrate the **frog** system to a group of prospective customers from Japan. Aside from the regular debugged version, there is a special version that works in Japanese. The developer decides to assign this special version a *version-name* of **:japanese**. The system is already loaded, so the developer invokes:

```
(make-system 'frog ':noload '(:update-directory :japanese))
```

To load this version in the future the developer must use the *version-name* argument to the **:version** keyword.

## 5.2 Using the **:version** and **:update-directory** keywords

This section shows how the user and the developer might apply the **:version** and **:update-directory** keywords to (1) update the **rodent** system from Release 4 to Release 5 and (2) be able to maintain multiple versions of **rodent** in parallel. In this example the site has a working version of the **rodent** system that runs in Release 4.4.

1. In order to use the **:version** and **:update-directory** features introduced in Release 4.5, the system developer must compile the system in Release 4.5 by an invocation of **make-system**; for example:

```
(make-system 'rodent ':compile)
```

To make this newly compiled version of **rodent** the released version, the system developer updates the system version-directory file appropriately:

```
(make-system 'rodent ':noload '(:update-directory :released))
```

Note: Assume that version #34 corresponds to Release 4.5 software.

2. The system developer wants to work with Release 5 software. To run **rodent** the developer must bring up a version of the system in Release 5 and recompile it. (Although recompilation is not necessarily required to move from one major software release to another, you *must* recompile the **rodent** system to update to Release 5.) Note: Assume that this recompilation in Release 5 created version #35.
3. A user running Release 4.5 can still load the released version of **rodent** by typing one of the following.

```
(make-system 'rodent)
```

or

```
(make-system 'rodent '(:version 34.))
```

However, the user cannot access the system while running Release 4.4 because the format of the system version-directory file has changed incompatibly.

4. The developer wants to run the experimental version of **rodent**, the one brought up in Release 5. The developer loads the system by issuing one of the following:

```
(make-system 'rodent '(:version :latest))
```

or

```
(make-system 'rodent '(:version 35.))
```

The developer finds problems with **rodent**, fixes them, makes patches, and decides to make the system now running the released version. While running Release 5 in this world, the developer uses **:update-directory** to specify the version:

```
(make-system 'rodent '(:update-directory :released))
```

Because some users still want to use the Release 4.5 version of **rodent**, the developer decides to give this version a version-name of **:old-system**. The developer boots Release 4.5, loads **rodent**, and designates this loaded system as **:old-system**.

```
(make-system 'rodent '(:update-directory :old-system))
```

A user who wants to load the Release 4.5 version of **rodent** must do one of the following:

```
(make-system 'rodent '(:version :old-system))
```

or

```
(make-system 'rodent '(:version 34.))
```

A user who wants to load the released version of **rodent** must do one of the following:

```
(make-system 'rodent)
```

or

```
(make-system 'rodent '(:version 35.))
```

### 5.3 Maintaining Parallel Systems for the LM-2 and the 3600

Normally, when a system is compiled, the system version-directory file is automatically updated. However, maintaining parallel systems for LM-2s and 3600s raises a special problem. Because you must specify **:no-increment-patch** in the compilation for a second machine in order to keep the version number at the same level, you must also specify **:update-directory** to get the correction information for the second machine in the system version-directory file.

**Example:** Assuming the LM-2 is the second machine and that you have already compiled the system on the 3600, you need to compile the system for the LM-2:

```
(make-system 'george ':compile ':noload ':no-increment-patch)
```

You then need to load the newly compiled files to record the system version information in the database.

```
(make-system 'george '(:version :newest) ':update-directory)
```

### 5.4 Describing a System

**describe-system** is a useful function for finding information about a system.

**describe-system** *system-name* &key (*show-files* *t*) *Function*  
(*show-transformations* *t*)

Displays useful information about the system named *system-name*. This includes the name of the system source file, the system package default if any, and component systems. For a patchable system, **describe-system** displays the system version and status, a typical patch file name, the sites maintaining the system, and, if the user wants, a listing of patches. If **:show-files** is *t*, it displays the history of the files in the system. Other possible values are **nil** (do not show file history) and **:ask** (ask the user). If **:show-transformations** is *t*, it displays the transformations required to make the system. Other possible values are **nil** (do not display transformations) and **:ask** (ask the user).

For finding information about patchable systems only: See the section "Finding Out About Patchable Systems".



## 6. Adding New Keywords to make-system

**make-system** keywords are defined as functions on the **si:make-system-keyword** property of the keyword. The functions are called with no arguments. Some of the relevant variables they can use are:

- si:\*system-being-made\*** *Variable*  
The internal data structure that represents the system being made.
- si:\*make-system-forms-to-be-evald-before\*** *Variable*  
A list of forms that are evaluated before the transformations are performed.
- si:\*make-system-forms-to-be-evald-after\*** *Variable*  
A list of forms that are evaluated after the transformations have been performed.
- si:\*make-system-forms-to-be-evald-finally\*** *Variable*  
A list of forms that are evaluated after the body of **make-system** has completed. This differs from **si:\*make-system-forms-to-be-evald-after\*** in that these forms are evaluated outside of the "compiler context", which sometimes makes a difference.
- si:\*query-type\*** *Variable*  
Controls how questions are asked. Its normal value is **:normal**. **:noconfirm** means no questions are asked, and **:selective** asks a question for each individual file transformation.
- si:\*silent-p\*** *Variable*  
If **t**, no messages are displayed.
- si:\*batch-mode-p\*** *Variable*  
If **t**, **:batch** was specified.
- si:\*redo-all\*** *Variable*  
If **t**, all transformations are performed, regardless of the condition functions.
- si:\*top-level-transformations\*** *Variable*  
A list of the names of transformations that will be performed, such as (**:fasload :readfile**).
- si:\*file-transformation-function\*** *Variable*  
The actual function that gets called with the list of transformations that need to be performed. The default is **si:do-file-transformations**.

**si:define-make-system-special-variable** *name form &optional* *Special Form*  
*(defvar-p t)*

Causes the variable *name* to be bound to *form*, which is evaluated at **make-system** time, during the body of the call to **make-system**. This allows you to define new variables similar to those already existent. If you specify *defvar-p* as (or defaulted to) *t*, *name* is defined with **defvar**. It is not given an initial value. If *defvar-p* is specified as *nil*, *name* belongs to some other program and is not **defvared** here.

The following simple example adds a new keyword to **make-system** called **:just-warn**, which means that **fdefine** warnings regarding functions being overwritten should be displayed, but the user should not be queried. (See the function **fdefine**.)

```
(si:define-make-system-special-variable
 inhibit-fdefine-warnings inhibit-fdefine-warnings nil)
```

```
(defun (:just-warn si:make-system-keyword) ()
 (setq inhibit-fdefine-warnings ':just-warn))
```

(See the variable **inhibit-fdefine-warnings**.)

**make-system** keywords can have effect either directly when called or by pushing a form to be evaluated onto **si:\*make-system-forms-to-be-evaluated-after\*** or one of the other two similar lists. In general, the only useful thing to do is to set some special variable defined by **si:define-make-system-special-variable**.

In addition to the ones mentioned earlier in this section, user-defined transformations can have their behavior controlled by new special variables, which can be set by new keywords. For example, if you want to get at the list of transformations to be performed, the right way would be to set **si:\*file-transformation-function\*** to a new function, which then might call **si:do-file-transformations** with a possibly modified list. That is how the **:print-only** keyword works.

Remember that when you execute **make-system**, it adds the loaded system to the system version-directory file of patchable systems unless you specify certain keywords that explicitly suppress this action. For example, **:print-only** is among these keywords. Certain user-defined keywords — those that rebind **si:\*file-transformation-function\*** and then recursively call **make-system** — must also take into account this updating feature of **make-system**. The following code is assumed to be in the **si** package.

```
(defun (:print-only make-system-keyword) ()
 (no-update-directory) ;Suppresses updating
 (setq *file-transformation-function* 'print-file-transformations))
```

## 7. Adding New Options to defsystem

Options to **defsystem** are defined as macros on the **si:defsystem-macro** property of the option keyword. Such a macro can expand into an existing option or transformation, or it can have side effects and return **nil**. They can use several variables, but the only one of general interest is **si:\*system-being-defined\***.

**si:\*system-being-defined\*** *Variable*

The internal data structure representing the system that is currently being constructed.

**si:define-defsystem-special-variable** *name form* *Special Form*

Causes *form* to be evaluated and *name* to be bound to the result during the expansion of the **defsystem** special form. This allows you to define new variables similar to **si:\*system-being-defined\***.

**si:define-simple-transformation** *name function default-condition* *Special Form*

*input-file-types output-file-types* &optional  
*pretty-names* (*compile-like t*) (*load-like nil ll-p*)

This is the most convenient way to define a new simple transformation. For example,

```
(si:define-simple-transformation :compile si:compile-file-1
  si:file-newer-than-file-p
  (:lisp) (:bin))
```

*input-file-types* and *output-file-types* are how a transformation specifies how many input file names and output file names it should receive as arguments, in this case one of each. They also, obviously, specify the default canonical file type for these pathnames.

The **si:compile-file-1** function is mostly like **compile-file**, except for its interface to packages. It takes *input-file* and *output-file* arguments.

*pretty-names*, an optional argument, specifies how the transformation will be printed in messages to the user. It can be a list of the imperative ("Compile"), the present participle ("Compiling"), and the past participle ("compiled"). Note that the past participle is not capitalized, because it is not used at the beginning of a sentence. *pretty-names* can be just a string, which is taken to be the imperative, and the system will conjugate the participles itself. If *pretty-names* is omitted or **nil** it defaults to the name of the transformation.

*compile-like* and *load-like*, both optional arguments, specify when the transformation should be performed. Compile-like transformations are performed when the **:compile** keyword is given to **make-system**. Load-like

transformations are performed unless the **:noload** keyword is given to **make-system**. By default *compile-like* is **t** but *load-like* is **nil**.

Complex transformations are just defined as normal macro expansions, for example,

```
(defmacro (:compile-load si:defsystem-macro)
  (input &optional com-dep load-dep
         com-cond load-cond)
  '(:fasload (:compile ,input ,com-dep ,com-cond)
             ,load-dep ,load-cond))
```



## 8. More Esoteric Transformations

It is sometimes useful to specify a transformation upon which something else can depend, which is not performed by default, but rather only when requested because of that dependency. The transformation nevertheless occupies a specific place in the hierarchy. The **:skip defsystem** macro allows specifying a transformation of this type. For example, suppose a special compiler for the read table is not ordinarily loaded into the system; the compiled version should still be kept up to date, and it needs to be loaded if the read table ever needs to be recompiled.

```
(defsystem reader
  (:pathname-default "AI: LMIO;")
  (:package system-internals)
  (:module defs "RDDEFS")
  (:module reader "READ")
  (:module read-table-compiler "RTC")
  (:module read-table "RDTBL")
  (:compile-load defs)
  (:compile-load reader (:fasload defs))
  (:skip :fasload (:compile read-table-compiler))
  (:rtc-compile-load read-table (:fasload read-table-compiler)))
```

Assume that there is a complex transformation **:rtc-compile-load** that is like **:compile-load**, except that it is built on a transformation called something like **:rtc-compile**, which uses the read table compiler rather than the Lisp compiler. In the above system, then, if the **:rtc-compile** transformation is to be performed, the **:fasload** transformation must be done on **read-table-compiler** first; that is, the read table compiler must be loaded if the read table is to be recompiled. If you say (**make-system 'reader 'compile**), then the **:compile** transformation will still happen on the **read-table-compiler** module, compiling the read table compiler if necessary. But if you issue (**make-system 'reader**), the reader and the read table will be loaded, but the **:skip** keeps this from happening to the read table compiler.

So far nothing has been said about what can be given as a *condition* for a transformation, except for the default functions that check for a source file being newer than the binary, and so on. In general, any function that takes the same arguments as the transformation function (for example, **compile-file**) and returns **t** if the transformation needs to be performed, can be in this place as a symbol, including, for example, a closure.

To take an example, suppose a file contains **compile-flavor-methods** for a system and should therefore be recompiled if any of the flavor method definitions change. In this case, the condition function for compiling that file should return **t** if either the source of that file itself or any of the files that define the flavors have changed. This is the purpose of the **:compile-load-init** complex transformation. It is defined in the **si** package like this:

```

(defmacro (:compile-load-init defsystem-macro)
  (input add-dep &optional com-dep load-dep
    &aux function)
  (setq function (let-closed ((*additional-dependent-modules*
    (parse-module-components add-dep *system-being-defined*))
    'compile-load-init-condition))
    '(:fasload (:compile ,input ,com-dep ,function) ,load-dep))

(defun compile-load-init-condition (source-file binary-file)
  (or (file-newer-than-file-p source-file binary-file)
    (local-declare ((special *additional-dependent-modules*)
      (other-files-newer-than-file-p
        *additional-dependent-modules*
        binary-file))))))

```

The condition function generated when this macro is used returns **t** either if **file-newer-than-file-p** would do so with those arguments, or if any of the other files in **add-dep** (which presumably is a *module specification*) are newer than the compiled code file. Thus the file (or module) to which the **:compile-load-init** transformation applies will be compiled if it or any of the source files on which it depends has been changed, and will be loaded under the normal conditions. In most (but not all cases), **com-dep** would be a **:fasload** transformation of the same files as **add-dep** specifies, so that all the files on which this one depends would be loaded before compiling it.

## 9. Patch Facility

The patch facility allows a system maintainer to manage new releases of a large system and issue patches to correct bugs. It is designed to be used to maintain the Lisp Machine system itself as well as applications systems that are large enough to be loaded into a Lisp world and saved into a FEP file (disk partition on an LM-2).

When a system of programs is very large, it needs to be maintained; for example, often problems are found and need to be fixed, or other little changes need to be made. However, it takes a long time to load up all of the files that make up such a system, and so rather than having users load up all the files every time they want to use the system, usually the files just get loaded once into a Lisp world, and then the Lisp world is saved away in a FEP file (disk partition on the LM-2). Users then use this file (disk partition), and copies of it are distributed. The problem is that since the users do not load up the system every time they want to use it, they do not get all the latest changes.

The purpose of the patch system is to solve this problem. A *patch* file is a little file that, when loaded, updates the old version of the system into the new version of the system. Most often, patch files just contain new function definitions; old functions are redefined to perform their new contracts. When you want to use a system, you first use the Lisp environment saved on the disk, and then you load all the latest patches. Patch files are very small, so loading them does not take much time. You can even load the saved environment, load up the latest patches, and then save it away, to save future users the trouble of even loading the patches. (Of course, new patches can be made later, and then these will have to be loaded if you want to get the very latest version.)

Every system has a series of patches that have been made to that system. To get the latest version of the system, you load each patch file in the series, in order. Sooner or later, the system maintainer will want to stop building more and more patches, and recompile everything, starting afresh. A complete recompilation is also necessary when a system is changed in a far-reaching way, in a way that cannot be done with a small patch. For example, if you completely reorganize a program or change a lot of names or conventions, you might need to completely recompile it to make it work again. After a complete recompilation, the old patch files are no longer suitable to use; loading them might even break things.

To keep track of all these changes the patch facility labels each version of a system with a two-part number. The two parts are called the *major version number* and the *minor version number*. The minor version number is increased every time a new patch is made; the patch is identified by the major and minor version number together. The major version number is increased when the program is completely recompiled, and at that time the minor version number is reset to zero. A complete system version is identified by the major version number, followed by a dot, followed by the minor version number.

The following typical scenario should clarify this.

1. A new system is created; its initial version number is 1.0.
2. Then a patch file is created; the version of the program that results from loading the first patch file into version 1.0 is called 1.1.
3. Then another patch file might be created, and loading that patch file into system 1.1 creates version 1.2.
4. Then the entire system is recompiled, creating version 2.0 from scratch.
5. Now the two patch files are irrelevant, because they fix old software; the changes that they reflect are integrated into system 2.0.

Note that the second patch file should only be loaded into system 1.1 in order to create system 1.2; you should not load it into 1.0 or any other system besides 1.1. It is important that all the patch files be loaded in the proper order, for two reasons.

- First, it is very useful that any system numbered 1.1 be exactly the same software as any other system numbered 1.1, so that if somebody reports a bug in version 1.1, it is clear just which software is being cited.
- Secondly, one patch might patch another patch; loading them in some other order might have the wrong effect.

The patch facility keeps track, in the file system, of all the patch files that exist, remembering which version each one creates. A separate numbered sequence of patch files exists for each major version of each system, for example, `lmfs-37-15.lisp`, `lmfs-37-16.lisp`, and so forth. All of them are stored in the file system, and the patch facility keeps track of where they all reside.

In addition to the patch files themselves, the *patch-directory file* contains the patch facility's database by which the patch facility keeps track of what minor versions exist for a major version, and what the last major version of a system is. For example, `lmfs-37.patch-dir` contains a listing of the patches made for major version 37 and a comment on why each patch was made. These files and how to make them are described in this section.

In order to use the patch facility, you must define your system with `defsystem` and declare it as patchable with the `:patchable` option. (See the section "Defining a System".) When you load your system with `make-system`, it is added to the list of all systems present in the world. (See the function `make-system`.) Whenever you use `make-system` to compile your patchable system, its major version in the file system is incremented; thus a major version is associated with a set of compiled code files.

The patch facility keeps track of which version of each patchable system is present, and where the data about that system reside in the file system. This information can be used to update the Lisp world automatically to the latest versions of all the systems it contains. Once a system is present, you can ask for the latest patches to

be loaded, ask which patches are already loaded, and add new patches. You can also load patches or whole new systems and then save the entire Lisp environment away in a FEP file (disk partition). See the function **disk-save**.

## 9.1 Finding Out About Patchable Systems

When a Lisp Machine is booted, it displays a line of information telling you what systems are present, and which version of each system is loaded. This information is returned by the function **si:system-version-info**. It is followed by a text string containing any additional information that was specified by whoever created the current world load (disk partition on the LM-2). See the function **disk-save**.

**print-system-modifications** &rest *system-names* Function

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

If **print-system-modifications** is called with arguments, only the modifications to *systems-named* are listed.

**si:get-system-version** &optional (*system* "System") Function

Returns three values. The first two are the major and minor version numbers of the version of *system* currently loaded into the machine. The third is the status of the system, as a keyword symbol: **:experimental**, **:released**, **:obsolete**, or **:broken**. *system* defaults to **System**. This returns **nil** if that system is not present at all.

**si:system-version-info** &optional (*brief-p* nil) Function

Returns a string giving information about which systems and what versions of the systems are loaded into the machine (for systems that differ from the released versions) and what microcode version is running. A typical string for it to produce is:

"System 242.264, Zmail 83.42, LMFS 37.31, Vision 10.23, Tape 21.9,  
microcode TMC5-MIC 264, FEP 17"

If *brief-p* is **t**, it uses short names, suppresses the microcode version, any systems that should not appear in the disk label comment, the name **System**, and the commas:

"242.264 Vis 10.23"

*nil*  
"Release 5.2,  
FEP 24"  
*t*  
"Rel 5.2"

**si:patch-loaded-p** *major-version minor-version &optional (system "System")* *Function*

A predicate that tells whether the loaded version of *system* is past (or at) the specified patch level. Returns **t** if:

- the major version loaded is *major-version* and the minor version loaded is greater than or equal to *minor-version*
- the major version loaded is greater than *major-version*

Otherwise, the function returns **nil**.

Releases have numbers and status associated with them, just as systems do. Symbolics staff assign the release number.

**si:get-release-version** *Function*

**si:get-release-version** returns three values, the release numbers and the status of the current world load:

- 5 Major version number
  - > Patch version number or string describing minor patch level
- Status of the world load as a keyword symbol:
- ✓ **:experimental**
  - :released**
  - :obsolete**
  - :broken**
  - nil** (when status cannot be determined)

## 9.2 Types of Patch Files

The patch facility maintains several different types of files in the directory associated with your system:

- System version-directory file
- Patch directory file
- Individual patch file

This directory is specified to **defsystem** via either the **:patchable** option or the **:pathname-default** option. These files are maintained automatically, but so that you will know what they are and when they are obsolete (because they are associated with an obsolete version of your system), they are described in this section.

System version information is recorded in a database called the *system version-directory file* for each patchable system. Whenever you run **make-system** it creates or updates this file, recording the name, type, and file version number of all

constituent files of each version of a patchable system. In addition, it contains keywords describing the status of the system (for example, **:released** and **:latest**), associating particular system versions with these keywords. (**make-system** automatically updates the file when you specify the **:update-directory** keyword: See the section "**make-system** keywords".) System version information is maintained in parallel for both LM-2 and 3600 systems.

The physical file type of the system version-directory file is shown for some host systems:

| <i>Host</i> | <i>File type</i>       |
|-------------|------------------------|
| TOPS-20     | PATCH-DIR              |
| UNIX        | pd                     |
| VMS         | VPD                    |
| ITS         | (PDIR)                 |
| LMFS        | patch-dir or directory |
| Multics     | patch-dir              |

Example: The system version-directory file for the **lmfs** system is:

```
q:>sys>lmfs>patch>lmfs.patch-dir.44
```

The host, device, and directory in this example come from the system definition.

The major benefit of this detailed record keeping is that your site can support multiple versions of the same system. General users and system developers can load specific versions of systems and specific versions of system files, even when newer and possibly incompatible versions have been made. Some examples:

- System developers can work on the *latest* versions of systems, editing and recompiling some files, without forcing the average user to contend with new and experimental changes to the system.
- General users, on the other hand, can load the stable, *released* versions.
- Symbolics can more easily distribute versions of the system other than the newest version.
- You can use pre-Release-5.0 versions of systems after recompiled versions have been made for Release 5.0.

In addition, you can load a system in several different ways:

- by version number
- by version name
- by designation as released, latest, or newest

To load a specific system, use the **:version** keyword: See the section "**make-system** keywords".

The released version is the fully debugged version intended for general use. You must explicitly update the system version-directory file to indicate that a system is released. See the **:update-directory** keyword to **make-system**.

The latest version is the most recently compiled version of the system. The system version-directory file is automatically updated whenever you compile or recompile the system.

The newest version is the version consisting of the most recently compiled version of each *file* of a system. The newest version differs from the latest version when individual files have been compiled by hand. The newest version of a system has no version number. Note that you cannot define patches for the newest system.

Each major version of the system has a *patch directory file*, which describes the individual patches for a particular major version.

Example: The patch directory file for major version #37 of the *lmfs* system:

```
q:>sys>lmfs>patch>lmfs-37.patch-dir.69
```

**make-system** creates a new patch directory file automatically when you recompile a system or use the **:increment-patch** option. See the section "**make-system** keywords".

Each minor version of the system has a patch source file, whose name is the system name, the major version number, the minor version number, .lisp file type, and the source file version number.

Example: The first patch of major version 37 of the *lmfs* system has the following pathname:

```
q:>sys>lmfs>patch>lmfs-37-1.lisp.1
```

Patch files get compiled, so you will find patch files like the following:

```
q:>sys>lmfs>patch>lmfs-37-1.bin.1
```

A slightly different set of file name conventions are used, if the **:patchable** option to **defsystem** is given an argument, telling it to put the patch files in a different directory than the one which holds the other files of the system.

On TOPS-20, the file names take these forms:

|                               |                                                                    |
|-------------------------------|--------------------------------------------------------------------|
| System version-directory file | EE:PS:<PATDIR> <i>system</i> .PATCH-DIRECTORY                      |
| Patch directory file          | EE:PS:<PATDIR> <i>system-<i>nnn</i></i> .PATCH-DIRECTORY           |
| Individual patch file         | EE:PS:<PATDIR> <i>system-<i>nnn-<i>mmm</i></i></i> .LISP (or .BIN) |

These file name conventions allow the patches for multiple systems to coexist in the same directory.



### 9.3 Loading Patches

#### **load-patches** &rest *options*

*Function*

Used to bring the current world up to the latest minor version of whichever major version it is, for all systems present, or for certain specified systems. If there are any patches available, **load-patches** offers to read them in. With no arguments, **load-patches** updates all the systems present in this world.

Note: When you do a **make-system** of a patchable system, **make-system** calls **load-patches** after loading the system. If **make-system** is silent, then **load-patches** is silent; if **make-system** asks for confirmation, then **load-patches** asks for confirmation.

**load-patches** returns **t** if any patches were made, and **nil** otherwise.

*options*, if supplied, is one or more keywords or system names. The following options are accepted:

- :systems list** *list* is a list of names of systems (symbols or strings) to be brought up to date. If this option is not specified, all systems are processed.
- :verbose** Prints an explanation of what is being done. This is the default.
- :selective** For each patch, says what it is and then asks you whether or not to load it. This is the default. Answering P turns off selective mode for any remaining patches to the current system.
- :noselective** Turns off **:selective**.
- :silent** Turns off both **:selective** and **:verbose**. In **:silent** mode all necessary patches are loaded without printing anything and without querying the user.
- :nowarn** Suppresses questions requiring operator response.
- system-name* The name of a system (symbol or string) to be brought up to date.

**load-patches** returns **t** if any patches were loaded, otherwise **nil**.

#### **load-and-save-patches** &rest *keyword-args*

*Function*

**load-and-save-patches** disables network services and MORE processing. If no one is logged in, it logs in anonymously. It loads any patches that need to be loaded and any new versions of the site files, calling **load-patches** with arguments of **:noselective** and any other keywords provided as *keyword-args*. If any patches have been loaded, it then calls **disk-save** to save the resulting world load. If no patches have been loaded, it restores network services to

their state before **load-and-save-patches** was called, and if it has logged in anonymously it logs out.

Before disk-saving on the 3600, **load-and-save-patches** prompts for the name of a FEP file in which to save the world load. Before disk-saving on the LM-2, it calls **print-disk-label** and prints an estimate of the size of the world load before prompting for a band in which to save the world load.

Call **load-and-save-patches** before you log in in order to avoid putting the contents of your init file into the saved world load.

## 9.4 Making Patches

During a typical maintenance session you might make several edits to a system's source files. The patch facility allows you to copy these edits into a patch file so that they can be automatically incorporated into the system to create a new minor version. Edits in a patch file can be of varying levels of complexity — modified function definitions, new functions, modified **defvars** and **defconsts**, or arbitrary forms to be evaluated, even including **loads** of new files.

Start Patch (m-x) and Start Private Patch (m-x) are two commands for initiating a patch.

**Start Patch (m-x)** Starts a new patch but does not move any Lisp forms into the patch file. Prompts you for the system you want to patch; it must be a system currently loaded. It allocates a new minor version number for that particular system and starts constructing the patch file in an editor buffer.

While you are making your patch file, the minor version number that has been allocated for you is reserved so that nobody else can use it. Thus, if two people are patching a system at the same time, they cannot both get the same minor version number. Also note that you can put together patches for only one system at a time.

If you do a subsequent patch after finishing the current patch (see **Finish Patch (m-x)**), **Start Patch (m-x)** asks you which system you wish to patch and start a new minor version.

**Start Private Patch (m-x)**

Similar to **Start Patch (m-x)**, but it does not have any relationship to systems, major and minor version numbers, and official patch directories. Instead of prompting for a system, it prompts for a file name. You can use other patching commands, like **Add Patch (m-x)**, **Finish Patch (m-x)**, and **Abort Patch (m-x)**. When you finish the patch it is written out to the specified file.

This command allows you to make a private patch file that you can load, test, and share with other users before you install a numbered patch that all users automatically get.

If you change a function, you should recompile it and test it; then, once it works, use Add Patch (m-X), Add Patch Changed Definitions (m-X), or Add Patch Changed Definitions of Buffer (m-X) to put the code in the patch file.

**Add Patch (m-X)** Adds the region (if there is one) or else the current definition to the patch file currently being constructed. If you mistakenly use the command on code that does not work, select the buffer containing the patch file and delete it. Then later you can use Add Patch (m-X) on the corrected version.

**Add Patch Changed Definitions of Buffer (m-X)** Selects each definition that was changed in the buffer and asks you whether or not you want the definition patched.

For each definition, you can respond as follows:

| <i>Response</i> | <i>Action</i>                                                                                               |
|-----------------|-------------------------------------------------------------------------------------------------------------|
| Y               | Patches the definition.                                                                                     |
| N               | Skips the definition.                                                                                       |
| P               | Patches the definition and any additional definitions in the same buffer without asking any more questions. |

A definition needs to be patched if it has been changed since it was last patched or if it has not been patched since the file was read into the buffer.

Note that patching any region of text lying entirely within a definition (with Add Patch (m-X)) counts as patching that definition.

**Add Patch Changed Definitions (m-X)** Selects a buffer in which definitions were changed and asks whether or not you want to patch the changed definitions. Answering N skips the buffer and proceeds to the next buffer, if any. Answering Y selects each definition that has changed in that buffer and asks you whether or not you want the definition patched. For each definition, you can respond as follows:

| <i>Response</i> | <i>Action</i>           |
|-----------------|-------------------------|
| Y               | Patches the definition. |

- N Skips the definition.
- P Patches the definition and any additional definitions in the same buffer without asking any more questions; when done, it proceeds to the next buffer.

If there are more buffers containing definitions to be patched, it asks questions again when it gets to the next buffer.

A definition needs to be patched if it has been changed since it was last patched or if it has not been patched since the file was read into the buffer.

Note that patching any region of text lying entirely within a definition (with Add Patch (n-X)) counts as patching that definition.

After making and testing all of your patches, use Finish Patch (n-X).

**Finish Patch (n-X)** Installs the patch file so that other users can load it. This compiles the patch file if you have not done so yourself (patches are always compiled). It prompts you for a comment describing the reason for the patch; **load-patches** and **print-system-modifications** print these comments.

Sometimes you start making a patch file and for a variety of reasons do not install it — for example, you decide to abort the patch, you need to end your work session at this machine, or your machine crashes.

**Abort Patch (n-X)** Deallocates the minor version number that was assigned by Start Patch (n-X). It tells Zmacs that you are no longer currently making a patch; the next time you do Start Patch (n-X), Zmacs starts a new patch instead of appending to the one in progress.

**Resume Patch (n-X)** Allows you to go back to a patch that you were not able to finish in the same session in which you started it. This command works only if you have previously saved all modified buffers.

If the system crashes, use Resume Patch (n-X) and then Abort Patch (n-X). Begin the patch again.

## 9.5 Patchable System Status

The patch system has the concept of the *status* of a major version of a system. The status is displayed with the system version, in places such as the system print herald and the **comment** properties in FEP files (disk partition comments on the LM-2 disk label). This status announces the state, or condition, of system software — for example, whether a system is released or still experimental.

Use **set-system-status** to change the status of a system.

The status is one of the following keywords:

### **:experimental**

The system has been built but has not yet been fully debugged and released to users. This is the default status when a new major version is created, unless it is overridden with the **:initial-status** option to **defsystem**.

### **:released**

The system is released for general use. This status produces no extra text in the print herald and the **comment** properties in FEP files.

### **:obsolete**

The system is no longer supported.

### **:broken**

Similar to **:experimental** but is used when the system was thought incorrectly to have been debugged and hence was assigned **:released** status.

**set-system-status** *system new-status* &optional *major-version* *only-update-on-disk-p* *Function*

Changes the status of a system.

*system*            The name of the system.

*major-version*    The number of the major version to be changed; if unsupplied it defaults to the version currently loaded into the Lisp world.

*new-status*        A defined keyword — **:experimental**, **:released**, **:obsolete**, **:broken**.

### *only-update-on-disk-p*

If its value of is **t**, the patch directory file is updated to show *new-status*, but the running Lisp environment is not modified.

Call **set-system-status** manually; you should *not* place it in patch files.



## Index

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                 |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|
| <p><b>3</b></p> <p>Maintaining Parallel Systems for the LM-2 and the</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | <p><b>3</b></p> <p>3600 19</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | <p><b>3</b></p> |
| <p><b>A</b></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | <p><b>A</b></p> <p>Finding Out</p> <p>Abort Patch (m-X) Zmacs command 34</p> <p>About Patchable Systems 29</p> <p>Add Patch (m-X) Zmacs command 34</p> <p>Add region to patch file 34</p> <p>Adding New Keywords to <b>make-system</b> 21</p> <p>Adding New Options to <b>defsystem</b> 23</p>                                                                                                                                                                                                                | <p><b>A</b></p> |
| <p><b>B</b></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | <p><b>B</b></p> <p><b>si:</b> <b>:batch</b> option for <b>make-system</b> 13</p> <p><b>si:</b> <b>*batch-mode-p*</b> variable 21</p> <p><b>:broken</b> system status 37</p> <p><b>:bug-reports</b> option for <b>defsystem</b> 3</p>                                                                                                                                                                                                                                                                          | <p><b>B</b></p> |
| <p><b>C</b></p> <p>Abort Patch (m-X) Zmacs command 34</p> <p>Add Patch (m-X) Zmacs command 34</p> <p>Finish Patch (m-X) Zmacs command 34</p> <p>Resume Patch (m-X) Zmacs command 34</p> <p>Start Patch (m-X) Zmacs command 34</p> <p>Start Private Patch (m-X) Zmacs command 34</p> <p><b>:compile</b> option for <b>make-system</b> 13</p> <p><b>:compile</b> transformation 7</p> <p>Compile-like transformations 23</p> <p><b>:compile-load</b> transformation 7</p> <p><b>:compile-load-init</b> transformation 7</p> <p>Complex transformations 23</p> <p><b>:component-systems</b> option for <b>defsystem</b> 3</p> <p>Conditions 7, 25</p> | <p><b>C</b></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | <p><b>C</b></p> |
| <p><b>D</b></p> <p>Loading the System</p> <p><b>:bug-reports</b> option for</p> <p><b>:component-systems</b> option for</p> <p><b>:initial-status</b> option for</p> <p><b>:maintaining-sites</b> option for</p> <p><b>:module</b> option for</p> <p><b>:name</b> option for</p> <p><b>:not-in-disk-label</b> option for</p>                                                                                                                                                                                                                                                                                                                       | <p><b>D</b></p> <p><b>si:</b> <b>define-defsystem-special-variable</b> special form 23</p> <p><b>si:</b> <b>define-make-system-special-variable</b> special form 22</p> <p><b>si:</b> <b>define-simple-transformation</b> special form 23</p> <p>Defining a System 3</p> <p>Definition 11</p> <p><b>defsystem</b> 3</p> <p><b>defsystem</b> 3</p> <p><b>defsystem</b> 3</p> <p><b>defsystem</b> 3</p> <p><b>defsystem</b> 3</p> <p><b>defsystem</b> 3</p> <p><b>defsystem</b> 3</p> <p><b>defsystem</b> 3</p> | <p><b>D</b></p> |

**:package** option for **defsystem** 3  
**:patchable** option for **defsystem** 3, 27, 30  
**:pathname-default** option for **defsystem** 3  
**:short-name** option for **defsystem** 3  
 Adding New Options to **defsystem** 23  
                           **defsystem** special form 3, 27  
                           Dependencies 7  
                           **describe-system** function 19  
                           Describing a System 19  
                           directory file 30  
                           **:do-components** transformation 7

## E

## E

## E

More Esoteric Transformations 25  
       **:experimental** system status 37

## F

## F

## F

Patch Facility 27  
 System facility 1  
       **:fasload** transformation 7  
       file 34  
       file 34  
       file 27  
       file 30  
       file 30  
       File types of patch files 30  
       **\*file-transformation-function\*** variable 21  
       files 30  
       Files 30  
       Finding Out About Patchable Systems 29  
       Finish Patch (m-X) Zmacs command 34  
       form 3, 27  
       form 23  
       form 22  
       form 23  
       **describe-system** function 19  
       **load-and-save-patches** function 33  
       **load-patches** function 33  
       **make-system** function 11, 13  
       **print-system-modifications** function 29  
       **set-system-status** function 37  
       **si:get-release-version** function 30  
       **si:get-system-version** function 29  
       **si:patch-loaded-p** function 30  
       **si:set-system-source-file** function 11  
       **si:system-version-info** function 29

## G

## G

## G

**si: get-release-version** function 30  
**si: get-system-version** function 29





**:version** option for  
 Adding New Keywords to  
**:latest** symbol in  
**:newest** symbol in  
**:released** symbol in  
  
**si:** **\*make-system-forms-to-be-evald-after\***  
           variable 21  
**si:** **\*make-system-forms-to-be-evald-before\***  
           variable 21  
**si:** **\*make-system-forms-to-be-evald-finally\***  
           variable 21  
  
 Introduction to  
   **make-system** 13  
   **make-system** 21  
   **make-system :version** option 13  
   **make-system :version** option 13  
   **make-system :version** option 13  
   **make-system** function 11, 13  
   **make-system** keywords 13  
   Making a System 13  
   Making a System 1  
   Making Patches 34  
   Minor version 30  
   Minor version number 27, 34  
   Module 3  
   **:module** option for **defsystem** 3  
   More Esoteric Transformations 25

## N

## N

## N

Adding  
 Adding  
  
 Major version  
 Minor version  
  
**:name** option for **defsystem** 3  
 New Keywords to **make-system** 21  
 New Options to **defsystem** 23  
**:newest** symbol in **make-system :version**  
           option 13  
**:no-increment-patch** option for **make-system** 13  
**:noconfirm** option for **make-system** 13  
**:noload** option for **make-system** 13  
**:noop** option for **make-system** 13  
**:noselective** option for **load-patches** 33  
**:not-in-disk-label** option for **defsystem** 3  
**:nowarn** option for **load-patches** 33  
**:nowarn** option for **make-system** 13  
   number 27  
   number 27, 34

## O

## O

## O

**:latest** symbol in **make-system :version**  
**:newest** symbol in **make-system :version**  
**:released** symbol in **make-system :version**  
   **:bug-reports**  
**:component-systems**  
   **:initial-status**  
**:maintaining-sites**  
   **:module**  
   **:name**  
   **:not-in-disk-label**  
   **:package**  
   **:patchable**  
**:pathname-default**  
   **:short-name**  
   **:noselective**  
  
**:obsolete** system status 37  
   option 13  
   option 13  
   option 13  
   option for **defsystem** 3  
   option for **defsystem** 3  
   option for **defsystem** 3  
   option for **defsystem** 3  
   option for **defsystem** 3  
   option for **defsystem** 3  
   option for **defsystem** 3  
   option for **defsystem** 3  
   option for **defsystem** 3  
   option for **defsystem** 3, 27, 30  
   option for **defsystem** 3  
   option for **defsystem** 3  
   option for **load-patches** 33

|                            |                                |    |
|----------------------------|--------------------------------|----|
| <b>:nowarn</b>             | option for <b>load-patches</b> | 33 |
| <b>:selective</b>          | option for <b>load-patches</b> | 33 |
| <b>:silent</b>             | option for <b>load-patches</b> | 33 |
| <b>:systems</b>            | option for <b>load-patches</b> | 33 |
| <b>:verbose</b>            | option for <b>load-patches</b> | 33 |
| <b>:batch</b>              | option for <b>make-system</b>  | 13 |
| <b>:compile</b>            | option for <b>make-system</b>  | 13 |
| <b>:increment-patch</b>    | option for <b>make-system</b>  | 13 |
| <b>:no-increment-patch</b> | option for <b>make-system</b>  | 13 |
| <b>:noconfirm</b>          | option for <b>make-system</b>  | 13 |
| <b>:noload</b>             | option for <b>make-system</b>  | 13 |
| <b>:noop</b>               | option for <b>make-system</b>  | 13 |
| <b>:nowarn</b>             | option for <b>make-system</b>  | 13 |
| <b>:print-only</b>         | option for <b>make-system</b>  | 13 |
| <b>:recompile</b>          | option for <b>make-system</b>  | 13 |
| <b>:reload</b>             | option for <b>make-system</b>  | 13 |
| <b>:selective</b>          | option for <b>make-system</b>  | 13 |
| <b>:silent</b>             | option for <b>make-system</b>  | 13 |
| <b>:update-directory</b>   | option for <b>make-system</b>  | 13 |
| <b>:version</b>            | option for <b>make-system</b>  | 13 |
| Adding New                 | Options to <b>defsystem</b>    | 23 |
| Finding                    | Out About Patchable Systems    | 29 |

P

P

P

|                   |                                                      |           |
|-------------------|------------------------------------------------------|-----------|
| Maintaining       | <b>:package</b> option for <b>defsystem</b>          | 3         |
|                   | Parallel Systems for the LM-2 and the 3600           | 19        |
|                   | Patch                                                | 27        |
| Abort             | Patch (m-X) Zmacs command                            | 34        |
| Add               | Patch (m-X) Zmacs command                            | 34        |
| Finish            | Patch (m-X) Zmacs command                            | 34        |
| Resume            | Patch (m-X) Zmacs command                            | 34        |
| Start             | Patch (m-X) Zmacs command                            | 34        |
| Start Private     | Patch (m-X) Zmacs command                            | 34        |
|                   | Patch directory file                                 | 30        |
|                   | Patch Facility                                       | 27        |
|                   | Patch file                                           | 27        |
| Add region to     | patch file                                           | 34        |
| Install           | patch file                                           | 34        |
| File types of     | patch files                                          | 30        |
| Types of          | Patch Files                                          | 30        |
|                   | Patch level                                          | 30        |
| <b>sl:</b>        | <b>patch-loaded-p</b> function                       | 30        |
|                   | <b>:patchable</b> option for <b>defsystem</b>        | 3, 27, 30 |
| Status of a       | patchable system                                     | 37        |
|                   | Patchable System Status                              | 37        |
| Finding Out About | Patchable Systems                                    | 29        |
| Loading           | Patches                                              | 33        |
| Making            | Patches                                              | 34        |
|                   | <b>:pathname-default</b> option for <b>defsystem</b> | 3         |
|                   | <b>:print-only</b> option for <b>make-system</b>     | 13        |
|                   | <b>print-system-modifications</b> function           | 29        |
| Start             | Private Patch (m-X) Zmacs command                    | 34        |

## Q

## Q

si: **\*query-type\*** variable 21

## Q

## R

## R

**:readfile** transformation 7  
**:recompile** option for **make-system** 13  
 si: **\*redo-all\*** variable 21  
 Add region to patch file 34  
**:released** symbol in **make-system :version**  
 option 13  
**:released** system status 37  
**:reload** option for **make-system** 13  
 Resume Patch (m-X) Zmacs command 34

## R

## S

## S

**:selective** option for **load-patches** 33  
**:selective** option for **make-system** 13  
 si: **set-system-source-file** function 11  
**set-system-status** function 37  
**:short-name** option for **defsystem** 3  
 si: **\*batch-mode-p\*** variable 21  
 si: **\*file-transformation-function\*** variable 21  
 si: **\*make-system-forms-to-be-evald-after\***  
 variable 21  
 si: **\*make-system-forms-to-be-evald-before\***  
 variable 21  
 si: **\*make-system-forms-to-be-evald-finally\***  
 variable 21  
 si: **\*query-type\*** variable 21  
 si: **\*redo-all\*** variable 21  
 si: **\*silent-p\*** variable 21  
 si: **\*system-being-defined\*** variable 23  
 si: **\*system-being-made\*** variable 21  
 si: **\*top-level-transformations\*** variable 21  
 si: **define-defsystem-special-variable** special  
 form 23  
 si: **define-make-system-special-variable** special  
 form 22  
 si: **define-simple-transformation** special form 23  
 si: **get-release-version** function 30  
 si: **get-system-version** function 29  
 si: **patch-loaded-p** function 30  
 si: **set-system-source-file** function 11  
 si: **system-version-info** function 29  
**:silent** option for **load-patches** 33  
**:silent** option for **make-system** 13  
 si: **\*silent-p\*** variable 21  
**defsystem** special form 3, 27  
**si:define-defsystem-special-variable** special form 23  
**si:define-make-system-special-variable** special form 22  
**si:define-simple-transformation** special form 23  
**:broken** system Start Patch (m-X) Zmacs command 34  
**:experimental** system status 37  
 status 37

## S

|                             |                                                     |    |
|-----------------------------|-----------------------------------------------------|----|
| <b>:obsolete</b> system     | status                                              | 37 |
| <b>:released</b> system     | status                                              | 37 |
| Patchable System            | Status                                              | 37 |
|                             | Status of a patchable system                        | 37 |
| <b>:latest</b>              | symbol in <b>make-system</b> <b>:version</b> option | 13 |
| <b>:newest</b>              | symbol in <b>make-system</b> <b>:version</b> option | 13 |
| <b>:released</b>            | symbol in <b>make-system</b> <b>:version</b> option | 13 |
| Defining a                  | System                                              | 1  |
| Describing a                | System                                              | 3  |
| Introduction to Making a    | System                                              | 19 |
| Making a                    | System                                              | 1  |
| Status of a patchable       | System                                              | 13 |
| Loading the                 | system                                              | 37 |
|                             | System Definition                                   | 11 |
|                             | System facility                                     | 1  |
|                             | System maintenance                                  | 27 |
| <b>:broken</b>              | system status                                       | 37 |
| <b>:experimental</b>        | system status                                       | 37 |
| <b>:obsolete</b>            | system status                                       | 37 |
| <b>:released</b>            | system status                                       | 37 |
| Patchable                   | System Status                                       | 37 |
|                             | System version-directory file                       | 30 |
|                             | System versions                                     | 27 |
| <b>si:</b>                  | <b>*system-being-defined*</b> variable              | 23 |
| <b>si:</b>                  | <b>*system-being-made*</b> variable                 | 21 |
| <b>si:</b>                  | <b>system-version-info</b> function                 | 29 |
| Finding Out About Patchable | Systems                                             | 29 |
| Updating                    | systems                                             | 27 |
| Maintaining Parallel        | Systems for the LM-2 and the 3600                   | 19 |
|                             | <b>:systems</b> option for <b>load-patches</b>      | 33 |

T

T

T

|                           |                                             |        |
|---------------------------|---------------------------------------------|--------|
| <b>si:</b>                | <b>*top-level-transformations*</b> variable | 21     |
| <b>:compile</b>           | transformation                              | 7      |
| <b>:compile-load</b>      | transformation                              | 7      |
| <b>:compile-load-init</b> | transformation                              | 7      |
| <b>:do-components</b>     | transformation                              | 7      |
| <b>:fasload</b>           | transformation                              | 7      |
| <b>:readfile</b>          | transformation                              | 7      |
|                           | Transformations                             | 7      |
| Compile-like              | transformations                             | 23     |
| Complex                   | transformations                             | 23     |
| Load-like                 | transformations                             | 23     |
| More Esoteric             | Transformations                             | 25     |
| User-defined              | transformations                             | 21, 23 |
|                           | Types of Patch Files                        | 30     |
| File                      | types of patch files                        | 30     |

U

U

U

|                               |                                                        |        |
|-------------------------------|--------------------------------------------------------|--------|
| Using the <b>:version</b> and | <b>:update-directory</b> keywords                      | 17     |
|                               | <b>:update-directory</b> option for <b>make-system</b> | 13     |
|                               | Updating systems                                       | 27     |
|                               | User-defined transformations                           | 21, 23 |

## V

|                                                   |                                                       |        |
|---------------------------------------------------|-------------------------------------------------------|--------|
| <b>si:*batch-mode-p*</b>                          | variable                                              | 21     |
| <b>si:*file-transformation-function*</b>          | variable                                              | 21     |
| <b>si:*make-system-forms-to-be-evald-after*</b>   | variable                                              | 21     |
| <b>si:*make-system-forms-to-be-evald-before*</b>  | variable                                              | 21     |
| <b>si:*make-system-forms-to-be-evald-finally*</b> | variable                                              | 21     |
| <b>si:*query-type*</b>                            | variable                                              | 21     |
| <b>si:*redo-all*</b>                              | variable                                              | 21     |
| <b>si:*silent-p*</b>                              | variable                                              | 21     |
| <b>si:*system-being-defined*</b>                  | variable                                              | 23     |
| <b>si:*system-being-made*</b>                     | variable                                              | 21     |
| <b>si:*top-level-transformations*</b>             | variable                                              | 21     |
|                                                   | <b>:verbose</b> option for <b>load-patches</b>        | 33     |
| Major                                             | version                                               | 30     |
| Minor                                             | version                                               | 30     |
| Using the                                         | <b>:version</b> and <b>:update-directory</b> keywords | 17     |
| Major                                             | version number                                        | 27     |
| Minor                                             | version number                                        | 27, 34 |
| <b>:latest</b> symbol in <b>make-system</b>       | <b>:version</b> option                                | 13     |
| <b>:newest</b> symbol in <b>make-system</b>       | <b>:version</b> option                                | 13     |
| <b>:released</b> symbol in <b>make-system</b>     | <b>:version</b> option                                | 13     |
|                                                   | <b>:version</b> option for <b>make-system</b>         | 13     |
| System                                            | version-directory file                                | 30     |
| System                                            | versions                                              | 27     |

## V

## V

## Z

|                           |               |    |
|---------------------------|---------------|----|
|                           | <b>Z</b>      |    |
| Abort Patch (m-X)         | Zmacs command | 34 |
| Add Patch (m-X)           | Zmacs command | 34 |
| Finish Patch (m-X)        | Zmacs command | 34 |
| Resume Patch (m-X)        | Zmacs command | 34 |
| Start Patch (m-X)         | Zmacs command | 34 |
| Start Private Patch (m-X) | Zmacs command | 34 |

## Z

## Z

*symbolics*<sup>™</sup>

---

# **COMP** The Compiler

---

---

Cambridge, Massachusetts

---

# **The Compiler**

**# 995009**

**February 1984**

**This document corresponds to Release 5.0.**

This document was prepared by the Documentation Group of Symbolics, Inc.

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Symbolics is a trademark of Symbolics, Inc., Cambridge, Massachusetts.

Copyright © 1981, 1979, 1978 Massachusetts Institute of Technology.  
All rights reserved.

Enhancements copyright © 1984, 1983, 1982 Symbolics, Inc. of Cambridge,  
Massachusetts.

All rights reserved. Printed in USA.

This document may not be reproduced in whole or in part without the prior written consent of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1



## Table of Contents

|                                                                | Page      |
|----------------------------------------------------------------|-----------|
| <b>1. The Basic Operations of the Compiler</b>                 | <b>1</b>  |
| 1.1 File Types                                                 | 1         |
| <b>2. How to Invoke the Compiler</b>                           | <b>3</b>  |
| <b>3. Input to the Compiler</b>                                | <b>5</b>  |
| 3.1 declare and eval-when                                      | 6         |
| <b>4. Compiler Declarations</b>                                | <b>9</b>  |
| <b>5. Compiler Warnings Database</b>                           | <b>13</b> |
| <b>6. Controlling Compiler Warnings</b>                        | <b>15</b> |
| <b>7. Compiler Source-level Optimizers</b>                     | <b>19</b> |
| <b>8. Files That Must Be Compiled on the 3600 and the LM-2</b> | <b>21</b> |
| <b>9. Files That Maclisp Must Compile</b>                      | <b>23</b> |
| <b>10. Putting Data in Compiled Code Files</b>                 | <b>25</b> |
| <b>Index</b>                                                   | <b>27</b> |

## 1. The Basic Operations of the Compiler

The purpose of the Lisp compiler is to convert Lisp functions into programs in the Lisp Machine's instruction set. Compiled functions run more quickly and take up less storage than interpreted code. They are executed directly by the microcode.

Compiled functions are represented in Lisp by compiled code objects, which contain machine code as well as various other information. On the 3600 the printed representation of the object is as follows:

```
#<DTP-COMPILED-FUNCTION name address>
```

On the LM-2 compiled code objects are represented by FEFs (Function Entry Frames), whose printed representation is as follows:

```
#<DTP-FEF-POINTER address name>
```

The assembly language for the 3600 is very similar to that of the LM-2. If you want to understand the output of the LM-2 compiler: See the section "How to Read Assembly Language". For information on 3600 assembly language: See the section "Assembly Language on the 3600".

The compiler checks for errors and issues warnings regarding faulty syntax, typographical errors, unbound symbols, and the like. See the section "Controlling Compiler Warnings".

### 1.1 File Types

The results of the compiler are written to a file of canonical type **:bin** (**:qbin** on the LM-2). The actual file types for compiled-code files are host-dependent, of course.

The following table gives the file types of **:bin** and **:qbin** files respectively.

| <i>Host type</i> | <i>File type for compiled code files on the 3600</i> | <i>File type for compiled code files on the LM-2</i> |
|------------------|------------------------------------------------------|------------------------------------------------------|
| ITS              | BIN                                                  | QBIN                                                 |
| Lisp Machine     | bin                                                  | qbin                                                 |
| Multics          | BIN                                                  | qbin                                                 |
| TENEX            | BIN                                                  | QBIN                                                 |
| TOPS-20          | BIN                                                  | QBIN                                                 |
| UNIX             | bn                                                   | qb                                                   |
| VAX/VMS          | BIN                                                  | QBN                                                  |



## 2. How to Invoke the Compiler

You can invoke the compiler from the Lisp Machine in several ways.

- Use the function **compile** to compile an interpreted function in the Lisp environment.
- Use Zmacs editor commands to read Lisp code in an editor buffer and compile it.
- Use **compiler:compile-file** and related functions to translate source files into compiled code files.

Note 1: Loading the compiled code file is almost the same as reading in the source file, except that the functions defined in the file are defined as compiled functions instead of interpreted functions.

Note 2: Compiling code in a Zmacs buffer causes some side effects on the Lisp environment, whereas compiling a source file does not. For more information:

- See the section "Compiling Code in a Zmacs Buffer".
- See the section "Compiling and Loading a File".

For general information on compiling, evaluating, and loading code:

- See the document *Zmacs Manual*.
- See the section "Compiling and Evaluating Lisp".
- See the document *Program Development Help Facilities*.

On the 3600 you can compile as many processes as you want at one time.

On the LM-2 only one process at a time can use the compiler. Attempts to invoke the compiler while it is running produce a message like the following:

```
[10:29 Compiler in process ZMACS-WINDOWS: waiting for resources.]
```

This means that you tried to run the compiler in the zmacs-windows process, but some other process is running in the compiler and is holding the global compiler lock. If you want to do your compilation, select the process that is using the compiler and either abort it or wait for it to finish. Your process that produced the error then wakes up and proceeds. Otherwise, you can give up on the attempt that produced the error by using `c-ABORT` on that process.

**compile** *name* &optional *lambda-exp* *Function*  
*name* is a function spec. See the section "Functions". The compiler converts *lambda-exp*, if supplied, into a compiled code object, saves the lambda-expression as the **:previous-expr-definition** and **:previous-definition**

properties of *name* if it is a symbol, and changes *name*'s definition to be the compiled code object. See the function **fdefine**.

**uncompile** *function-spec*

*Function*

If *function-spec* is not defined as an interpreted function and it has a **:previous-expr-definition** property, then **uncompile** restores the function cell from the value of the property. (Otherwise, **uncompile** does nothing and returns "Not compiled".) This "undoes" the effect of **compile**. See the function **undefun**.

**compiler:compile-file** *infile* &optional *outfile in-package*  
*dont-set-default-p*

*Function*

The file *infile* is given to the compiler, and the output of the compiler is written to a file whose name is *infile* with a file type on the Lisp Machine of .bin for a 3600 (.qbin for an LM-2). For a description of the input format for files to the compiler: See the section "Input to the Compiler". *outfile* lets you change where the output is written. *dont-set-default-p* suppresses the changing of the default file name to *infile*, which normally occurs.

**compiler:compile-file-load** &rest *compile-file-args*

*Function*

**compiler:compile-file-load** compiles a file and then loads in the resulting compiled code file. Its arguments are the same as those of **compiler:compile-file**. See the function **compiler:compile-file**.

To examine a compiled function in symbolic form: See the function **disassemble**.

### 3. Input to the Compiler

The purpose of **compiler:compile-file** is to take a file and produce a translated version that does the same thing as the original except that the functions are compiled. **compiler:compile-file** reads through the input file, processing the forms in it one by one. For each form, suitable binary output is sent to the compiled code file, which when loaded reproduces the effect of that source form. The differences between source files and compiled code files are that:

1. The latter are in a compressed binary form that reads and executes much faster but cannot be edited.
2. Function definitions in compiled code files have been translated from Lisp forms to compiled code objects.

Thus, if the source contains a (**defun ...**) form at top level, then when the compiled code file is loaded, the function is defined as a compiled function. If, on the other hand, the source file contains a form that is not of a type known specially to the compiler, then that form (encoded in binary format) is output "directly" into the compiled code file, so that when that file is loaded that form is evaluated. For example, if the source file contains (**setq x 3**), then the compiler places in the compiled code file instructions to set **x** to **3** at load time (that is, when the compiled code file is loaded into the Lisp environment). (It happens that compiled code files have a specific way to **setq** a symbol. For a more general form, the compiled code file would contain instructions to recreate the list structure of a form and then call **eval** on it.)

Sometimes you might want to put things in the compiled code file that are not meant merely to be translated into binary form. Top-level macro definitions fall into this category; the macros must actually get defined within the compiler in order for the compiler to be able to expand them at compile time. So when a macro form is seen, it should (sometimes) be evaluated at compile time, and should (sometimes) be put into the compiled code file.

Compiler declarations also fall into this category. Compiler declarations are forms that should be evaluated at compile time in order to tell the compiler something. They should not be put into the compiled code file, unless they are useful for working incrementally on the functions in the file, compiling them one by one from the editor.

### 3.1 declare and eval-when

You might want the compiler to handle forms in a variety of ways. (See the section "Input to the Compiler".) You might want a form to be:

- Put into the compiled code file (compiled, of course), or not.
- Evaluated within the compiler, or not.
- Evaluated if the file is read directly into Lisp, or not.

The compiler recognizes two forms that allow you to tell it exactly what to do with a form: the completely general **eval-when** and the less general **declare**.

An **eval-when** form looks like this:

```
(eval-when times-list
          form1
          form2
          ...)
```

The *times-list* can contain one or more of the symbols **load**, **compile**, or **eval**.

*If this symbol is present Then forms are*

|                |                                                                                                                                                                                                |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>load</b>    | Written into the compiled code file to be evaluated when the compiled code file is loaded, with the exception that <b>defun</b> forms put the compiled definition into the compiled code file. |
| <b>compile</b> | Evaluated in the compiler.                                                                                                                                                                     |
| <b>eval</b>    | Evaluated when read into Lisp; this is because <b>eval-when</b> is defined as a special form in Lisp. (The compiler ignores <b>eval</b> in the <i>times-list</i> .)                            |

Example: The following form would define **foo** as a macro in the compiler and when the file is read in interpreted, but not when the compiled code file is loaded.

```
(eval-when (compile eval) (macro foo (x) (cadr x)))
```

Note: For the rest of this section, we use lists such as are given to **eval-when** (for example, **(load eval)**, **(load compile)**) to describe when forms are evaluated.

A **declare** form looks like:

```
(declare form1 form2 ...)
```

**declare** is defined in Lisp as a special form that does nothing, so the forms within a **declare** are not evaluated at **eval** time. The compiler does the following upon finding *form* within a **declare**: If *form* is a call to either **special** or **unspecial**, *form* is treated as **(load compile)**; otherwise it is treated as **(compile)**.

In addition to recognizing **declare** as the first forms in the body of a function, the compiler recognizes **declare** as the first forms in the bodies of the following:

|                 |                  |
|-----------------|------------------|
| <b>let</b>      | <b>let*</b>      |
| <b>do</b>       | <b>do*</b>       |
| <b>do-named</b> | <b>do*-named</b> |
| <b>prog</b>     | <b>prog*</b>     |
| <b>lambda</b>   |                  |

This means that you can have **special** declarations that are local to any of these blocks.

If a form is not enclosed in either an **eval-when** or a **declare**, then the times at which it will be evaluated depend on the form. The following table summarizes at what times evaluation will take place for any given form seen at top level by the compiler.

(eval-when *times-list form1 ...*)  
*times-list*

(declare (special ...)) or (declare (unspecial ...))  
**(load compile)**

(declare *anything-else*)  
**(compile)**

(special ...) or (unspecial ...)  
**(load compile eval)**

(macro ...) or (defmacro ...) or (defsubst ...)  
**(load compile eval)**

(comment ...)  
 Ignored at all times.

(compiler-let ((var val) ...) *body...*)  
 Processes the *body* in its normal fashion, but at **(compile eval)** time, the indicated variable bindings are in effect. These variables typically affect the operation of the compiler or of macros. See the section "Nesting Macros".

(local-declare (*decl decl ...*) *body...*)  
 Processes the *body* in its normal fashion, with the indicated declarations added to the front of the list that is the value of **local-declarations**.

(defflavor ...) or (defstruct ...)  
**(load compile eval)**

(defun ...) or (defmethod ...) or (defselect ...)  
**(load eval)**, but at load time what is processed is not this form itself, but the result of compiling it.

*anything-else*  
**(load eval)**



Sometimes a macro wants to return more than one form for the compiler top level to see (and to be evaluated). The following facility is provided for such macros. If the following form is seen at the compiler top level, all of the *forms* are processed as if they had been at compiler top level.

```
(progn (quote compile) form1 form2 ...)
```

(Of course, in the interpreter they will all be evaluated, and the **(quote compile)** will harmlessly evaluate to the symbol **compile** and be ignored.) For additional discussion: See the section "Macros Expanding Into Many Forms".

**eval-when** &quote *times* &rest *body*

*Special Form*

When seen by the interpreter, if one of the *times* is the symbol **eval**, then the *body* forms are evaluated. Otherwise **eval-when** does nothing; but when seen by the compiler, this special form does special things. See the section "Input to the Compiler".

**declare** &quote &rest *ignore*

*Special Form*

**declare** does nothing, and returns the symbol **declare**.

But when seen by the compiler, this special form does special things. See the section "Input to the Compiler". There is also a different use of **declare**, used in conjunction with the **arglist** function. See the function **arglist**. See the section "Compiler Declarations".

## 4. Compiler Declarations

This section describes functions meant to be called during compilation, and variables meant to be set or bound during compilation, by using **declare** or **local-declare**.

**local-declare** *declarations &body body*

*Special Form*

A **local-declare** form looks like

```
(local-declare (decl1 decl2 ...)
  form1
  form2
  ...)
```

Example:

```
(local-declare ((special foo1 foo2))
  (defun larry ()
    )
  (defun george ()
    )
  ); end of local-declare
```

Each *decl* is consed onto the list **local-declarations** while the *forms* are being evaluated (in the interpreter) or compiled (in the compiler). There are two uses for this. First, it can be used to pass information from outer macros to inner macros. Secondly, the compiler will specially interpret certain *decls* as local declarations, which only apply to the compilations of the *forms*. It understands the following forms:

**(special sym1 sym2 ...)**

The variables *sym1*, *sym2*, and so on are treated as special variables during the compilation of the *forms*.

**(unspecial sym1 sym2 ...)**

The variables *sym1*, *sym2*, and so on are treated as local variables during the compilation of the *forms*.

**(arglist . arglist)**

Putting this local declaration around a **defun** saves *arglist* as the argument list of the function, to be used instead of its lambda-list if anyone asks what its arguments are. This is purely documentation.

**(values . values)**

Putting this local declaration around a **defun** saves *values* as the return values list of the function, to be used if anyone asks what values it returns. This is purely documentation.

**(def function . defining-forms)**

*function* is defined for the compiler during the compilation of the *forms*. The compiler uses this to keep track of macros and open-codeable functions (**defsubst**s) defined in the file being compiled. Note that the **cddr** of this item is a function.

**special** &quote &rest *symbols* *Special Form*

Declares each of the *symbols* to be "special" for the compiler.

**unspecial** &quote &rest *symbols* *Special Form*

Removes any "special" declarations of the *symbols* for the compiler.

The next three declarations are primarily for Maclisp compatibility.

**\*expr** &quote &rest *functions* *Special Form*

Declares each function spec in the list of *functions* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined, which appears at the end of the compilation.

**\*lexpr** &quote &rest *functions* *Special Form*

Declares each function spec in the list of *functions* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined printed at the end of the compilation.

**\*fexpr** &quote &rest *functions* *Special Form*

Declares each function spec in the list of *functions* to be the name of a special form. In addition it prevents these names from appearing in the list of functions referenced but not defined printed at the end of the compilation.

The compile-time values of the following variables affect the operation of the compiler. You can set these variables by including in his file forms such as

```
(declare (setq open-code-map-switch t))
```

**run-in-maclisp-switch** *Variable*

This variable works only on the LM-2. If this variable is non-**nil**, the compiler tries to warn you about any constructs that do not work in Maclisp. By no means are all Lisp Machine system functions not built-in to Maclisp cause for warnings — only those which could not be written by the user in Maclisp (for example, **make-array**, **value-cell-location**, and so on). Also, lambda-list keywords such as **&optional** and initialized **prog** variables are mentioned. This switch also inhibits the warnings for obsolete Maclisp functions. The default value of this variable is **nil**.

**obsolete-function-warning-switch** *Variable*

If this variable is non-**nil**, the compiler tries to warn you whenever an "obsolete" Maclisp-compatibility function, such as **maknam** or **samepnamep**, is used. The default value is **t**.

**allow-variables-in-function-position-switch** *Variable*

This variable works only on the LM-2. If this variable is non-**nil**, the compiler allows the use of the name of a variable in function position to mean that the variable's value should be **funcall**'d. This is for compatibility with old Maclisp programs. The default value of this variable is **nil**.

**open-code-map-switch** *Variable*

If this variable is non-**nil**, the compiler attempts to produce inline code for the mapping functions (**mapc**, **mapcar**, and so on, but not **mapatoms**) if the function being mapped is an anonymous some lambda-expression. This allows that function to reference the local variables of the enclosing function without the need for special declarations. The generated code is also more efficient. The default value is **t**.

**all-special-switch** *Variable*

If this variable is non-**nil**, the compiler regards all variables as special, regardless of how they were declared. This provides compatibility with the interpreter at the cost of efficiency. The default is **nil**.

**inhibit-style-warnings-switch** *Variable*

If this variable is non-**nil**, all compiler style-checking is turned off. Style checking is used to issue obsolete function warnings and won't-run-in-Maclisp warnings, and other sorts of warnings. The default value is **nil**.

See the macro **inhibit-style-warnings**. The **inhibit-style-warnings** macro acts on only one level of an expression.

**compiler-let** *&quote bindlist &rest body* *Macro*

Syntactically identical to **let**, **compiler-let** allows compiler switches to be bound locally at compile time, during the processing of the *body* forms. Value forms are evaluated at compile time.

Example:

```
(compiler-let ((open-code-map-switch nil))  
  (map (function (lambda (x) ...)) foo))
```

This prevents the compiler from open-coding the **map**. When interpreted, **compiler-let** is equivalent to **let**. This is so that global switches that affect the behavior of macro expanders can be bound locally.

**compiler:compiler-verbose** *Variable*

The compiler displays a message (using **standard-output**) each time it starts compiling a function when the value is **t**. The default value is **nil**.



## 5. Compiler Warnings Database

Compiler warnings are kept in an internal database, and several functions and editor commands are provided that allow you to inspect and manipulate this database in various ways.

The database of compiler warnings is organized by pathname; warnings that were generated during the compilation of a particular file are kept together, and this body of warnings is identified by the generic pathname of the file being compiled. Any warnings that were generated while compiling some function not in any file (for example, by using the **compile** function on some interpreted code) are stored under the pathname **nil**. For each pathname, the database has entries, each of which associates the name of a function (or a flavor) with the warnings generated during its compilation.

The database starts out empty when you cold boot. Whenever you compile a file, buffer, or function, the warnings generated during its compilation are entered into the database. If you recompile a function, the old warnings are removed, and any new warnings are inserted. If you get some warnings, fix the mistakes, and recompile everything, the database becomes empty again.

Warnings are printed out as well as stored in the database. If the value of the special variable **suppress-compiler-warnings** is not **nil**, warnings are not printed, although they are still stored in the database.

The database has a printed representation. **print-compiler-warnings** produces this printed representation from the database, and **compiler:load-compile-warnings** updates the database from a saved printed representation. Following are the details:

**print-compiler-warnings** &optional *files* (*stream* *standard-output*) *Function*

Prints out the compiler warnings database. If *files* is **nil** (the default), it prints the entire database. Otherwise, *files* should be a list of generic pathnames, and only the warnings for the specified files are printed. (**nil** can be a member of the list, too, in which case warnings for functions not associated with any file are also printed.) The output is sent to *stream*; you could use this to send the results to a file.

**compiler:load-compiler-warnings** *file* &optional (*flush-old-warnings* *t*) *Function*

Updates the compiler warnings database. *file* should be the pathname of a file containing the printed representation of the compiler warnings related to the compilation of one or more files. If *flush-old-warnings* is **t** (the default), any existing warnings in the database for the files in question are completely replaced by the warnings in *file*. If *flush-old-warnings* is **nil**, the warnings in *file* are added to those already in the database.

The printed representation of a set of compiler warnings is sometimes stored in a file. You can create such a file using **print-compiler-warnings**, but it is usually created with **make-system** given the **:batch** option. The default type for such files is **CWARNS**.

Several Zmacs commands deal with the database.

#### Compiler Warnings (m-X)

Prints the compiler warnings database into a buffer called Compiler Warnings, creates the buffer if it does not exist already, and switches to that buffer. You can peruse the compiler warnings by scrolling around and doing text searches through them.

#### Edit Compiler Warnings (m-X)

Prompts you with the name of each file mentioned in the database, allowing you to edit the warnings for that file. It then splits the Zmacs frame into two windows: the upper window displays a warning message, and the lower one displays the source code whose compilation caused the warning. After you have finished editing each function, **c-** gets you to the next warning: the top window scrolls to show the next warning, and the bottom window displays the function associated with this warning. Successive **c-**s take you through all of the warning messages for all of the files you specified. When you are done, the last **c-** puts the frame back into its one-window configuration.

#### Edit File Warnings (m-X)

Asks you for the name of the file whose warnings you want to edit. You can give either the source file or the compiled file. Only warnings for this file are edited. If the database does not have any entries for the file you specify, the command prompts you for the name of a file that contains the warnings, in case you know that the warnings are stored in another file.

#### Load Compiler Warnings (m-X)

Prompts you for the name of a file containing the printed representation of some compiler warnings and loads them into the database. (This is like the **compiler:load-compiler-warnings** function.) This command always passes **t** as the *flush-old-warnings* argument; that is, it replaces the old warnings rather than merging with them. The default file type is **CWARNS** and the default version is **:newest** (the latest version).

## 6. Controlling Compiler Warnings

The compiler performs style checking on all forms. Style checking is implemented by the **compiler:style-checker** property on a symbol; the value of the property is called on all forms whose **car** is that symbol, except those immediately enclosed in **inhibit-style-warnings**.

By controlling the compile-time values of the variables **run-in-maclisp-switch**, **obsolete-function-warning-switch**, and **inhibit-style-warning-switch** you can enable or disable some of the warning messages of the compiler. (See the section "Compiler Declarations".)

The following special form is also useful:

**inhibit-style-warnings** *body* *Macro*

Prevents the compiler from performing style-checking on the top level of *body*. Style-checking will still be done on the arguments of *body*. Both obsolete function warnings and won't-run-in-Maclisp warnings are done by means of the style-checking mechanism, so, for example,

```
(setq bar (inhibit-style-warnings (value-cell-location foo)))
```

does not warn that **value-cell-location** will not work in Maclisp, but

```
(inhibit-style-warnings (setq bar (value-cell-location foo)))
```

will warn, since **inhibit-style-warnings** applies only to the top level of the form inside it (in this case, to the **setq**).

Sometimes functions take arguments that they deliberately do not use. Normally the compiler warns you if your program binds a variable that it never references. In order to disable this warning for variables that you know you are not going to use, you can do one of two things.

- You can name the variables **ignore** or **ignored**. The compiler will not complain if a variable by one of these names is not used. Furthermore, by special dispensation, it is all right to have more than one variable in a lambda-list that has one of these names.
- You can simply use the variable for effect (ignoring its value) at the front of the function. Example:

```
(defun the-function (list fraz-name fraz-size)
  fraz-size      ; This argument is not used.
  ...)
```

This has the advantage that **arglist** will return a more meaningful argument list for the function, rather than returning something with **ignores** in it. See the function **arglist**.



The compiler uses a set of variables and functions to keep track of which functions have been defined and which have been referenced. These are the basis for the messages "FOO was defined but never referenced" that occur during compiling.

The following variables, used for this purpose, are implemented as hash tables:

**sys:file-local-declarations**  
**compiler:functions-defined**  
**compiler:functions-referenced**

**compiler:function-defined** *fspec* *Function*  
**function-defined** tells the compiler that the function *fspec* has been defined (by putting it into the hash table in **compiler:functions-defined**).

**compiler:file-declare** *thing declaration value* *Function*  
**file-declare** enters a declaration in the table **sys:file-local-declarations** for the remaining extent of the compilation environment.

(compiler:file-declare 'foo 'special t)

**compiler:file-declaration** *thing declaration* *Function*  
**file-declaration** looks up a declaration in the table **sys:file-local-declarations**. It returns the declaration when *thing* is a declaration of type *declaration* and **nil** otherwise.

In addition to the above functions, **compiler:function-referenced** is useful for requesting compiler warnings in certain esoteric cases. Normally, the compiler notices whenever any function *x* uses (calls) any other function *y*; it takes note of all these uses, and then warns you at the end of the compilation if the function *y* got called but was neither defined nor declared (by **\*expr**). See the special form **\*expr**.

This usually does what you want, but sometimes the compiler has no way of telling that a certain function is being used. Suppose that instead of *x*'s containing any forms that call *y*, *x* simply stores *y* away in a data structure somewhere, and someplace else in the program that data structure is accessed and **funcall** is done on it. In this case the compiler cannot see that this is going to happen; the result is that it cannot note the function usage and hence cannot create a warning message. In order to make such warnings happen, you can explicitly call the function **compiler:function-referenced** at compile-time.

**compiler:function-referenced** *what* &optional *by* *Function*  
*compiler:default-warning-function*  
*what* is a symbol that is being used as a function. *by* can be any function spec. **compiler:function-referenced** must be called at compile-time while a compilation is in progress. It tells the compiler that the function *what* is referenced by *by*. When the compilation is finished, if the function *what* has not been defined, the compiler issues a warning to the effect that *by* referred to the function *what*, which was never defined.

**compiler:make-obsolete** *function reason**Special Form*

This special form declares a function to be obsolete; code that calls it gets a compiler warning, under the control of **obsolete-function-warning-switch**. See the function **obsolete-function-warning-switch**. This is used by the compiler to mark as obsolete some Maclisp functions that exist in Zetalisp but should not be used in new programs. It can also be useful when maintaining a large system, as a reminder that a function has become obsolete and that its use should be phased out. An example of an obsolete-function declaration is:

```
(compiler:make-obsolete create-mumblefrotz
  "use MUMBLIFY with the :FROTZ option instead")
```



## 7. Compiler Source-level Optimizers

The optimizer feature of the 3600 compiler works differently in the LM-2 compiler. The most important difference on the 3600 is that when an optimizer for a function (not for a special form) is run, the argument forms it sees have already been optimized.

An optimizer can be used to transform code into an equivalent but more efficient form, which can be compiled better. For example, (**eq** *obj* **nil**) is transformed into (**null** *obj*), which can be compiled better.

An optimizer can also be used to tell the compiler how to compile a special form. For example, in the interpreter **do** is a special form, implemented by a function that takes quoted arguments and calls **eval**. In the compiler, **do** is expanded in a macro-like way by an optimizer, into equivalent Lisp code using **prog**, **cond**, and **go**, which the compiler understands.

The compiler stores optimizers for source code on property lists in order to make it easy for you to add them. The compiler finds the optimizers to apply to a form by looking for the **compiler:optimizers** property of the symbol that is the **car** of the form. The value of this property should be a list of optimizers, each of which must be a function of one argument. The compiler tries each optimizer in turn, passing the form to be optimized as the argument. An optimizer that returns the original form unchanged (**eq** to the argument) has "done nothing", and the next optimizer is tried. If the optimizer returns anything else, it has "done something", and the whole process starts over again. Only after all the optimizers have been tried and have done nothing is an ordinary macro definition processed. This is so that the macro definitions, which will be seen by the interpreter, can be overridden for the compiler by optimizers.

Do not use optimizers to define new language features, because they take effect only in the compiler; the interpreter (that is, the evaluator) does not know about optimizers. So an optimizer should not change the effect of a form; it should produce another form that does the same thing, possibly faster or with less memory. If you want to actually change the form to do something else, you should use macros.

**compiler:add-optimizer** *&quote target-function optimizer-name* *Special Form*  
*&rest optimized-into...*

Puts *optimizer-name* on *target-function*'s optimizers list if it is not there already. *optimizer-name* is the name of an optimization function, and *target-function* is the name of the function calls that are to be processed. Neither is evaluated.

(**compiler:add-optimizer** *target-function optimizer-name optimize-into-1 optimize-into-2...* also remembers *optimize-into-1*, and so on, as names of

functions that can be called in place of *target-function* as a result of the optimization.

## 8. Files That Must Be Compiled on the 3600 and the LM-2

In some cases it will be necessary to conditionalize pieces of programs so that one version runs on the LM-2 and another runs on the 3600.

To facilitate this, the list returned by (**status features**) on the 3600 contains the Lisp object **3600** (as a fixnum, 3600 decimal), whereas on the LM-2 it does not. To conditionalize a piece of a program so that it runs on both the LM-2 and the 3600, use the **#+** conditional expressions.

Example: Suppose a function **solarize-screen** that on the LM-2 expects coordinate pairs of the form  $(x,y)$  was changed to expect them in  $(y,x)$  order on the 3600. One way to write machine-dependent code is to conditionalize it, as follows:

```
#+cadr (solarize-screen arg1 arg2) ;the LM-2 version
#+3600 (solarize-screen arg2 arg1) ;the 3600 version
```

For information on sharp-sign (**#**) abbreviations: See the section "Sharp-sign Abbreviations".



## 9. Files That Maclisp Must Compile

Certain programs are intended to be run both in Maclisp and in Zetalisp. Their source files need some special conventions. For example, all **special** declarations must be enclosed in **declares**, so that the Maclisp compiler sees them. The main issue is that many functions and special forms of Zetalisp do not exist in Maclisp. It is suggested that you turn on **run-in-maclisp-switch** in such files, which warns you about a great many problems that your program might have if you try to run it in Maclisp.

The macro-character combination "**#Q**" causes the object that follows it to be visible only when compiling for Zetalisp. The combination "**#M**" causes the following object to be visible only when compiling for Maclisp. These work both on subexpressions of the objects in the file, and at top level in the file. To conditionalize top-level objects, however, it is better to put the macros **if-for-lispm** and **if-for-maclisp** around them. (You can only put these around a single object.) The **if-for-lispm** macro turns off **run-in-maclisp-switch** within its object, preventing spurious warnings from the compiler. The **#Q** macro-character cannot do this, since it can be used to conditionalize any Lisp object, not just a top-level form.

To allow a file to detect what environment it is being compiled in, the following macros are provided:

**if-for-lispm** &rest *forms* *Macro*

Seen at the top level of the compiler, *forms* is passed to the compiler top level if the output of the compiler is a compiled code file intended for Zetalisp. If the Zetalisp interpreter sees this it evaluates *forms* (the macro expands into *forms*).

**if-for-maclisp** &rest *forms* *Macro*

Seen at the top level of the compiler, *forms* is passed to the compiler top level if the output of the compiler is a compiled code file intended for Maclisp (for example, if the compiler is COMPLR). If the Zetalisp interpreter sees this it ignores it (the macro expands into **nil**).

**if-for-maclisp-else-lispm** *maclisp-form lispm-form* *Macro*

When (**if-for-maclisp-else-lispm** *form1 form2*) is seen at the top level of the compiler, *form1* is passed to the compiler top level if the output of the compiler is a compiled code file intended for Maclisp; otherwise *form2* is passed to the compiler top level.

**if-in-lispm** &rest *forms* *Macro*

In Zetalisp, (**if-in-lispm** *forms*) causes *forms* to be evaluated; in Maclisp, *forms* is ignored.



**if-in-maclisp** &rest *forms**Macro*

In Maclisp, (**if-in-maclisp** *forms*) causes *forms* to be evaluated; in Zetalisp, *forms* is ignored.

When you have two definitions of one function, one conditionalized for one machine and one for the other, put them next to each other in the source file with the second "**(defun)**" indented by one space, and the editor will put both function definitions on the screen when you ask to edit that function.

In order to make sure that those macros are defined when reading the file into the Maclisp compiler, you must make the file start with a prelude, which should look like:

```
(declare (cond ((not (status feature lispm))  
                (load '|AI: LISP2; CONDIT|))))
```

This will do nothing when you compile the program on the Lisp Machine. If you compile it with the Maclisp compiler, it will load in definitions of the above macros, so that they will be available to your program. The form (**status feature lispm**) is generally useful in other ways; it evaluates to **t** when evaluated on the Lisp Machine and to **nil** when evaluated in Maclisp.

## 10. Putting Data in Compiled Code Files

It is possible to make a compiled code file containing data, rather than a compiled program. This can be useful to speed up loading of a data structure into the machine, as compared with reading in printed representations. Also, certain data structures, such as arrays, do not have a convenient printed representation as text, but can be saved in compiled code files.

In compiled programs, the constants are saved in the compiled code file in this way. The compiler optimizes by making constants which are `equal` become `eq` when the file is loaded. This does not happen when you make a data file yourself; identity of objects is preserved. Note that when a compiled code file is loaded, objects that were `eq` when the file was written are still `eq`; this does not normally happen with text files.

The following types of objects can be represented in compiled code files:

- Symbols
- Numbers of all kinds
- Lists
- Strings
- Arrays of all kinds
- Instances (for example, hash tables)
- Compiled code objects

When an instance is put (dumped) into a compiled code file, it is sent a `:fasd-form` message, which must return a Lisp form which, when evaluated, will recreate the equivalent of that instance. This is because instances are often part of a large data structure, and simply dumping all of the instance variables and making a new instance with those same values is unlikely to work. Instances remain `eq`; the `:fasd-form` message is sent only the first time a particular instance is encountered during writing of a compiled code file. If the instance does not accept the `:fasd-form` message, it cannot be dumped.

**`sys:dump-forms-to-file`** *file forms-list &optional attribute-list* *Function*

`sys:dump-forms-to-file` writes data to a file in binary form. *forms-list* is a list of Lisp forms, each of which is dumped in sequence. It dumps the forms, not their results. The forms are evaluated when you load the file.

For example, suppose `a` is a variable bound to any Lisp object, such as a list or array. The following example creates a compiled code file that recreates the variable `a` with the same value:

```
(sys:dump-forms-to-file "f:>foo>aval"  
  (list '(setq a ',a)))
```

For the purposes of understanding what this function does, you can consider that it is the same as the following:

```
(defun sys:dump-forms-to-file (file forms)
  (with-open-file (s file ':direction ':output)
    (dolist (f forms)
      (print f s))))
```

The real definition writes a binary file so it will load faster. It can also dump arrays, which you cannot write to a Lisp source file.

*attribute-list* supplies an optional attribute list for the resulting compiled code file. It has basically the same result when loading the binary file as the file attribute list does for **compiler:compile-file**. Its most important application is for controlling the package that the file is loaded into.

```
(sys:dump-forms-to-file "foo" forms-list '(:package "user"))
```

**sys:dump-forms-to-file** always puts a package attribute into the binary file it writes. If you do not specify the *attribute-list* argument, or if *attribute-list* does not contain a **:package** attribute, the function uses the **user** package. This is to ensure that package prefixes on symbols are always interpreted when they are loaded as they were intended when the file was dumped.

To examine a compiled code file, use **si:unbin-file** (**si:unfasl** on the LM-2). The output format from **unbin-file** is similar to that of **unfasl** but is improved to include disassembled code for any compiled functions in the compiled code file.

**si:unbin-file** *file* &optional *outfile* *Function*

Converts the compiled code file *file* on the 3600 to human-readable file, which you can optionally specify. It includes disassembled code for any compiled functions in the compiled code file.

**si:unfasl** *input-file* &optional *output-file* *Function*

Converts the compiled code file *input-file* on the LM-2 to a human-readable file, which you can optionally specify.

## Index

#

#

#

# + 3600 21  
 # + cadr 21  
 #M macro-character 23  
 #Q macro-character 23

3

3

3

Files That Must Be Compiled on the 3600 and the LM-2 21

A

A

A

**compiler:** **add-optimizer** special form 19  
**all-special-switch** variable 11  
**allow-variables-in-function-position-switch**  
 variable 11  
 ignored arguments 15  
 Arrays in compiled code files 25

B

B

B

The Basic Operations of the Compiler 1  
 BIN file type 1

C

C

C

Conditionalizing code 21  
 Arrays in compiled code files 25  
 Compiled code objects in compiled code files 25  
 File type for compiled code files 1  
 Instances in compiled code files 25  
 Lists in compiled code files 25  
 Numbers in compiled code files 25  
 Putting Data in Compiled Code Files 25  
 Symbols in compiled code files 25  
 Compiled code objects 1  
 Compiled code objects in compiled code files 25  
 Compiler Warnings (m-X) Zmacs command 13  
 Edit Compiler Warnings (m-X) Zmacs command 13  
 Edit File Warnings (m-X) Zmacs command 13  
 Load Compiler Warnings (m-X) Zmacs command 13  
 Files That Maclisp Must Compile 23  
**compiler:** **compile** function 3  
**compiler:** **compile-file** function 4  
**compiler:** **compile-file-load** function 4  
 Arrays in compiled code files 25  
 Compiled code objects in compiled code files 25  
 File type for compiled code files 1  
 Instances in compiled code files 25  
 Lists in compiled code files 25  
 Numbers in compiled code files 25

|                             |                                                    |
|-----------------------------|----------------------------------------------------|
| Putting Data in Symbols In  | Compiled Code Files 25                             |
|                             | compiled code files 25                             |
|                             | Compiled code objects 1                            |
| Files That Must Be          | Compiled code objects in compiled code files 25    |
|                             | Compiled on the 3600 and the LM-2 21               |
|                             | Compiler 26                                        |
| How to Invoke the           | Compiler 3                                         |
| Input to the                | Compiler 5                                         |
| Optimizer feature of the    | compiler 19                                        |
| The Basic Operations of the | Compiler 1                                         |
| Warnings from the           | compiler 15                                        |
|                             | Compiler declarations 5, 9                         |
| Specifying                  | compiler environments 23                           |
|                             | Compiler not reentrant 3                           |
|                             | Compiler Source-level Optimizers 19                |
|                             | Compiler variables 15                              |
| Controlling                 | Compiler Warnings 15                               |
|                             | Compiler Warnings (m-X) Zmacs command 13           |
| Edit                        | Compiler Warnings (m-X) Zmacs command 13           |
| Load                        | Compiler Warnings (m-X) Zmacs command 13           |
|                             | Compiler Warnings Database 13                      |
| Print                       | compiler warnings database 13                      |
| Update                      | compiler warnings database 13                      |
|                             | <b>compiler-let</b> macro 11                       |
| <b>compiler:</b>            | <b>compiler-verbose</b> variable 11                |
|                             | <b>compiler:add-optimizer</b> special form 19      |
|                             | <b>compiler:compile-file</b> function 4            |
|                             | <b>compiler:compile-file-load</b> function 4       |
|                             | <b>compiler:compiler-verbose</b> variable 11       |
|                             | <b>compiler:file-declaration</b> function 16       |
|                             | <b>compiler:file-declare</b> function 16           |
|                             | <b>compiler:function-defined</b> function 16       |
|                             | <b>compiler:function-referenced</b> function 16    |
|                             | <b>compiler:functions-defined</b> variable 15      |
|                             | <b>compiler:functions-referenced</b> variable 15   |
|                             | <b>compiler:load-compiler-warnings</b> function 13 |
|                             | <b>compiler:make-obsolete</b> special form 17      |
|                             | Conditionalizing code 21                           |
|                             | Controlling Compiler Warnings 15                   |
|                             | Controlling warning messages 15                    |

## D

|                          |                                       |
|--------------------------|---------------------------------------|
| Putting                  | Data in Compiled Code Files 25        |
| Compiler Warnings        | Database 13                           |
| Print compiler warnings  | database 13                           |
| Update compiler warnings | database 13                           |
| Compiler                 | Declaration 9                         |
|                          | declarations 5, 9                     |
|                          | <b>declare</b> 6                      |
|                          | <b>declare</b> and <b>eval-when</b> 6 |
|                          | <b>declare</b> special form 6, 8      |
|                          | Definitions, functions 23             |
|                          | <b>defun</b> special form 23          |
|                          | <b>do</b> special form 6              |
|                          | <b>do*</b> special form 6             |
|                          | <b>do*-named</b> special form 6       |
|                          | <b>do-named</b> special form 6        |

## D

## D

DTP-COMPILED-FUNCTION 1  
**sys:** **dump-forms-to-file** function 25

**E**

Zwei  
 Function  
 Specifying compiler  
**declare** and

**lambda**

**E**

Edit Compiler Warnings (m-X) Zmacs command 13  
 Edit File Warnings (m-X) Zmacs command 13  
 editor 3  
 Entry Frame 1  
 environments 23  
**eval-when** 6  
**eval-when** special form 6, 8  
 Evaluation 5  
 Evaluation of special forms 6  
**\*expr** special form 10  
 expression 6

**E****F**

Optimizer  
 Source  
 BIN  
 QBIN  
 Edit  
**compiler:**  
**compiler:**  
**sys:**  
 Arrays in compiled code  
 Compiled code objects in compiled code  
 File type for compiled code  
 Instances in compiled code  
 Lists in compiled code  
 Numbers in compiled code  
 Putting Data in Compiled Code  
 Symbols in compiled code

**compiler:add-optimizer** special  
**compiler:make-obsolete** special  
**declare** special  
**defun** special  
**do** special  
**do\*** special  
**do\*-named** special  
**do-named** special  
**eval-when** special  
**\*expr** special  
**\*fexpr** special  
**let** special  
**let\*** special  
**\*lexpr** special

**F**

**:fasd-form** message 25  
 Fasdump 25  
 feature of the compiler 19  
 FEF 1  
**\*fexpr** special form 10  
 file 5  
 file type 1  
 file type 1  
 File type for compiled code files 1  
 File Types 1  
 File Warnings (m-X) Zmacs command 13  
**file-declaration** function 16  
**file-declare** function 16  
**file-local-declarations** variable 15  
 files 25  
 files 25  
 files 1  
 files 25  
 files 25  
 files 25  
 Files 25  
 files 25  
 Files That Maclisp Must Compile 23  
 Files That Must Be Compiled on the 3600 and the  
 LM-2 21  
 form 19  
 form 17  
 form 6, 8  
 form 23  
 form 6  
 form 6  
 form 6  
 form 6  
 form 6  
 form 6, 8  
 form 10  
 form 10  
 form 6  
 form 6  
 form 10

**F**

|                                        |                             |             |
|----------------------------------------|-----------------------------|-------------|
| <b>local-declare</b>                   | special form                | 9           |
| <b>prog</b>                            | special form                | 6           |
| <b>prog*</b>                           | special form                | 6           |
| <b>special</b>                         | special form                | 10          |
| <b>unspecial</b>                       | special form                | 10          |
| Evaluation of special                  | forms                       | 6           |
| Function Entry                         | Frame                       | 1           |
| <b>compile</b>                         | function                    | 3           |
| <b>compiler:compile-file</b>           | function                    | 4           |
| <b>compiler:compile-file-load</b>      | function                    | 4           |
| <b>compiler:file-declaration</b>       | function                    | 16          |
| <b>compiler:file-declare</b>           | function                    | 16          |
| <b>compiler:function-defined</b>       | function                    | 16          |
| <b>compiler:function-referenced</b>    | function                    | 16          |
| <b>compiler:load-compiler-warnings</b> | function                    | 13          |
| <b>print-compiler-warnings</b>         | function                    | 13          |
| <b>si:unbin-file</b>                   | function                    | 26          |
| <b>si:unfasl</b>                       | function                    | 26          |
| <b>sys:dump-forms-to-file</b>          | function                    | 25          |
| <b>uncompile</b>                       | function                    | 4           |
|                                        | Function Entry              | Frame 1     |
| <b>compiler:</b>                       | <b>function-defined</b>     | function 16 |
| <b>compiler:</b>                       | <b>function-referenced</b>  | function 16 |
| Definitions,                           | functions                   | 23          |
| <b>compiler:</b>                       | <b>functions-defined</b>    | variable 15 |
| <b>compiler:</b>                       | <b>functions-referenced</b> | variable 15 |

H

H

H

How to Invoke the Compiler 3

I

I

I

|                                      |          |    |
|--------------------------------------|----------|----|
| <b>if-for-lisp</b>                   | macro    | 23 |
| <b>if-for-maclisp</b>                | macro    | 23 |
| <b>if-for-maclisp-else-lisp</b>      | macro    | 23 |
| <b>if-in-lisp</b>                    | macro    | 23 |
| <b>if-in-maclisp</b>                 | macro    | 24 |
| <b>ignore</b>                        | variable | 15 |
| Ignored arguments                    |          | 15 |
| <b>inhibit-style-warnings</b>        | macro    | 15 |
| <b>inhibit-style-warnings-switch</b> | variable | 11 |
| Input to the Compiler                |          | 5  |
| Instances in compiled code files     |          | 25 |
| How to Invoke the Compiler           |          | 3  |

How to

L

L

L

Files That Must Be Compiled on the 3600 and the

**compiler:**

|                                            |              |    |
|--------------------------------------------|--------------|----|
| <b>lambda</b>                              | expression   | 6  |
| <b>let</b>                                 | special form | 6  |
| <b>let*</b>                                | special form | 6  |
| <b>*lexpr</b>                              | special form | 10 |
| Lists in compiled code files               |              | 25 |
| LM-2                                       |              | 21 |
| Load Compiler Warnings (m-X) Zmacs command |              | 13 |
| <b>load-compiler-warnings</b>              | function     | 13 |
| <b>local-declare</b>                       | special form | 9  |

M

M

M

Compiler Warnings (m-X) Zmacs command 13  
Edit Compiler Warnings (m-X) Zmacs command 13  
Edit File Warnings (m-X) Zmacs command 13  
Load Compiler Warnings (m-X) Zmacs command 13  
Maclisp 10  
Files That  
  **compiler-let** macro 11  
  **if-for-lisp** macro 23  
  **if-for-maclisp** macro 23  
**if-for-maclisp-else-lisp** macro 23  
  **if-in-lisp** macro 23  
  **if-in-maclisp** macro 24  
**Inhibit-style-warnings** macro 15  
  **#M** macro-character 23  
  **#Q** macro-character 23  
  **compiler:** **make-obsolete** special form 17  
  **.fasd-form** message 25  
Controlling warning messages 15  
Files That Must Be Compiled on the 3600 and the LM-2 21  
Files That Maclisp Must Compile 23

N

N

N

Compiler not reentrant 3  
Numbers in compiled code files 25

O

O

O

Compiled code objects 1  
Compiled code objects in compiled code files 25  
  **obsolete-function-warning-switch** variable 10  
  **open-code-map-switch** variable 11  
The Basic Operations of the Compiler 1  
Optimizer feature of the compiler 19  
Compiler Source-level Optimizers 19

P

P

P

Print compiler warnings database 13  
**print-compiler-warnings** function 13  
**prog** special form 6  
**prog\*** special form 6  
Putting Data in Compiled Code Files 25

Q

Q

Q

QBIN file type 1

R

R

R

Compiler not reentrant 3  
**run-in-maclisp-switch** variable 10, 23



## S

**Compiler**  
**compiler:add-optimizer** special form 19  
**compiler:make-obsolete** special form 17  
**declare** special form 6, 8  
**defun** special form 23  
**do** special form 6  
**do\*** special form 6  
**do\*-named** special form 6  
**do-named** special form 6  
**eval-when** special form 6, 8  
**\*expr** special form 10  
**\*fexpr** special form 10  
**let** special form 6  
**let\*** special form 6  
**\*lexpr** special form 10  
**local-declare** special form 9  
**prog** special form 6  
**prog\*** special form 6  
**special** special form 10  
**unspecial** special form 10  
 Evaluation of special forms 6  
**special** special form 10  
 Special variables 6, 11  
 Specifying compiler environments 23  
 switches 10  
 Symbols in compiled code files 25  
**sys:dump-forms-to-file** function 25  
**sys:file-local-declarations** variable 15  
 Warning

## S

## S

## T

Files That Maclisp Must Compile 23  
 Files That Must Be Compiled on the 3600 and the LM-2 21  
 BIN file type 1  
 QBIN file type 1  
 File type for compiled code files 1  
 File Types 1

## T

## T

## U

**si: unbin-file** function 26  
**uncompile** function 4  
**si: unfasl** function 26  
**unspecial** special form 10  
 Update compiler warnings database 13

## U

## U

## V

|                                                    |           |        |
|----------------------------------------------------|-----------|--------|
| <b>all-special-switch</b>                          | variable  | 11     |
| <b>allow-variables-in-function-position-switch</b> | variable  | 11     |
| <b>compiler:compiler-verbose</b>                   | variable  | 11     |
| <b>compiler:functions-defined</b>                  | variable  | 15     |
| <b>compiler:functions-referenced</b>               | variable  | 15     |
| <b>ignore</b>                                      | variable  | 15     |
| <b>inhibit-style-warnings-switch</b>               | variable  | 11     |
| <b>obsolete-function-warning-switch</b>            | variable  | 10     |
| <b>open-code-map-switch</b>                        | variable  | 11     |
| <b>run-in-maclisp-switch</b>                       | variable  | 10, 23 |
| <b>sys.file-local-declarations</b>                 | variable  | 15     |
| Compiler                                           | variables | 15     |
| Special                                            | variables | 6, 11  |

## V

## V

## W

|  |                             |                              |    |
|--|-----------------------------|------------------------------|----|
|  | <b>Controlling</b>          | warning messages             | 15 |
|  |                             | Warning switches             | 10 |
|  | <b>Controlling Compiler</b> | Warnings                     | 15 |
|  | Compiler                    | Warnings (m-X) Zmacs command | 13 |
|  | Edit Compiler               | Warnings (m-X) Zmacs command | 13 |
|  | Edit File                   | Warnings (m-X) Zmacs command | 13 |
|  | Load Compiler               | Warnings (m-X) Zmacs command | 13 |
|  | Compiler                    | Warnings Database            | 13 |
|  | Print compiler              | warnings database            | 13 |
|  | Update compiler             | warnings database            | 13 |
|  |                             | Warnings from the compiler   | 15 |

## W

## W

## Z

|  |                                     |               |    |
|--|-------------------------------------|---------------|----|
|  | <b>Compiler Warnings (m-X)</b>      | Zmacs command | 13 |
|  | <b>Edit Compiler Warnings (m-X)</b> | Zmacs command | 13 |
|  | <b>Edit File Warnings (m-X)</b>     | Zmacs command | 13 |
|  | <b>Load Compiler Warnings (m-X)</b> | Zmacs command | 13 |
|  |                                     | Zwei editor   | 3  |

## Z

## Z

# MISCT Other Tools

---

---

## **Other Tools**

**# 995014**

**February 1984**

**This document corresponds to Release 5.0.**

This document was prepared by the Documentation Group of Symbolics, Inc.

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Symbolics is a trademark of Symbolics, Inc., Cambridge, Massachusetts.

Copyright © 1984, 1981, Symbolics, Inc. of Cambridge, Massachusetts.

All rights reserved. Printed in USA.

This document may not be reproduced in whole or in part without the prior written consent of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

## Table of Contents

|                                                    | Page     |
|----------------------------------------------------|----------|
| <b>1. The Inspector</b>                            | <b>1</b> |
| 1.1 Introduction to the Inspector                  | 1        |
| 1.2 The Inspector History Pane                     | 1        |
| 1.3 The Inspector Inspection Pane                  | 3        |
| 1.3.1 Inspection Pane Display                      | 3        |
| 1.4 The Inspector Interaction Pane                 | 4        |
| 1.5 Special Characters Recognized by the Inspector | 4        |
| 1.6 The Inspector Menu Pane                        | 5        |
| <b>2. Flavor Examiner</b>                          | <b>7</b> |
| <b>Index</b>                                       | <b>9</b> |



## List of Figures

Figure 1. The Inspector.

2

## 1. The Inspector

### 1.1 Introduction to the Inspector

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 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.

The Inspector can be part of another program or it can be used standalone; for example, the Display Debugger can utilize some of the panes of the Inspector. Note, however, that although the display looks the same as that of the standalone Inspector, the handling of the mouse buttons depends upon the particular program being run.

You can enter the standalone Inspector via [Inspect] in the System menu or by the **inspect** function, which inspects its argument, if any.

See the document *Program Development Tools and Techniques*.

Figure 1 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)

### 1.2 The Inspector History Pane

The history pane maintains a list of all objects that have been inspected. The last recently displayed object is at the top of the list, and the most recently displayed object is at the bottom.

You can inspect any mouse-sensitive object in the history pane by clicking on it. In addition, you can perform other operations by placing the mouse cursor in the *line region*, which is the left-hand side of the history pane, the area bounded by the margin on one side and the list of objects on the other. In the line region the shape of the mouse cursor changes to a rightward-pointing arrow.

- Clicking left in the line region inspects the object. This is sometimes useful



Figure 1. The Inspector.

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|
| <pre> #&lt;Package GLOBAL 20315016&gt; 1130 "GLOBAL" SI:PKG-NEW-SYMBOL-EXTERNAL-ONLY →:SOURCE-FILE-NAME </pre>                                                                                                                                                                                                                                                                                                                                                                                                        | <pre> Exit Return Modify DeCache Clear Set \ </pre> |
| <p style="text-align: center;"><i>More above</i></p> <p style="text-align: center;"><i>More below</i></p> <p style="text-align: center;"><i>Top of object</i></p> <pre> SI:PKG-NEW-SYMBOL-EXTERNAL-ONLY Value is unbound Function is #'SI:PKG-NEW-SYMBOL-EXTERNAL-ONLY Property list: (:SOURCE-FILE-NAME #&lt;LOGICAL-PATHNAME "SYS: SYS; PACKAGE"&gt;) Package: #&lt;Package SYSTEM-INTERNALS 20043232&gt; </pre> <p style="text-align: center;"><i>Bottom of object</i></p>                                         |                                                     |
| <p style="text-align: center;"><i>Top of object</i></p> <pre> :SOURCE-FILE-NAME Value is :SOURCE-FILE-NAME Function is unbound Property list: NIL Package: #&lt;Package KEYWORD 20333021&gt; </pre> <p style="text-align: center;"><i>Bottom of object</i></p>                                                                                                                                                                                                                                                        |                                                     |
| <p style="text-align: center;"><i>Top of object</i></p> <pre> #&lt;LOGICAL-PATHNAME "SYS: SYS; PACKAGE"&gt; An instance of FS:LOGICAL-PATHNAME. #&lt;Message handler for FS:LOGICAL-PATHNAME&gt; FS:HOST: #&lt;LOGICAL-HOST SYS&gt; FS:DEVICE: :UNSPECIFIC FS:DIRECTORY: ("SYS") FS:NAME: "PACKAGE" FS:TYPE: NIL FS:VERSION: NIL SI:PROPERTY-LIST: #&lt;LMFS-PATHNAME "Q:&gt;sys&gt;sys&gt;package"&gt; FS:STRING-FOR-PRINTING: "SYS: SYS; PACKAGE" </pre> <p style="text-align: center;"><i>Bottom of object</i></p> |                                                     |
| <pre> : Inspect the indicated object. M: Remove the indicated object. 12/07/83 19:10:49 sr USER: Ty1 Console idle 10 minutes </pre>                                                                                                                                                                                                                                                                                                                                                                                   |                                                     |

when the object is a list and it is inconvenient to position the mouse at the open parenthesis.

- Clicking middle deletes the object from the history.

The history pane also maintains a cache allowing quick redisplay of previously displayed objects. This means that merely reinspecting an object does not reflect any changes in its state. Clicking middle in the line region deletes the object from the cache as well as deleting it from the history pane. Use [DeCache] in the menu pane to clear everything from the cache.

The history pane has a scroll bar at the far left, as well as scrolling zones in the middle of its top and bottom edges. The last three lines of the history are always the objects being inspected in the inspection panes.

### 1.3 The Inspector Inspection Pane

Each inspection pane can inspect a different object. When you inspect an object it appears in the large inspection pane at the bottom, and the previously inspected objects shift upward.

At the top of an inspection pane is either a label, which is the printed representation of the object being inspected in that window, or the words "a list", which means a list is being inspected. The main body of an inspection pane is a display of the components of the object, labelled with their names, if any. You can scroll this display using the scroll bar on the left or the "more above" and "more below" scrolling zones at the top and bottom.

Clicking on any mouse-sensitive object in an inspection pane inspects that object. The three mouse buttons have distinct meanings, however.

- Clicking left inspects the object in the bottom pane, pushing the previous objects up.
- Clicking middle inspects the object but leaves the source (namely, the object being inspected in the window in which the mouse was clicked) in the second pane from the bottom.
- Clicking right tries to find and inspect the function associated with the selected object (for example, the function binding if a symbol was selected).

#### 1.3.1 Inspection Pane Display

The inspection display that is chosen depends upon the type of the object:

|        |                                                                                                                                     |
|--------|-------------------------------------------------------------------------------------------------------------------------------------|
| Symbol | The name, value, function, property list, and package of the symbol are displayed. All but the name and the package are modifiable. |
|--------|-------------------------------------------------------------------------------------------------------------------------------------|

---

|                      |                                                                                                                                                                                                               |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| List                 | The list is displayed ground by the system grinder. Any piece of substructure is selectable, and any car or atom in the list can be modified.                                                                 |
| Instance             | The flavor of the instance, the method table, and the names and values of the instance-variable slots are displayed. The instance-variables are modifiable.                                                   |
| Closure, Entity      | The function, and the names and values of the closed variables are displayed. For an entity the type or class is displayed as well. The values of the closed variables are modifiable.                        |
| Named structure      | The names and values of the slots are displayed. The values are modifiable.                                                                                                                                   |
| Array                | The leader of the array is displayed if present. For one-dimensional arrays, the elements of the array are also displayed. The elements are modifiable.                                                       |
| Compiled code object | The disassembled code is displayed.                                                                                                                                                                           |
| Select Method        | The keyword/function pairs are shown, in alphabetical order by keyword. The function associated with a keyword is settable via the keyword.                                                                   |
| Stack Frame          | This is a special internal type used by the Display Debugger. It is displayed as either interpreted code (a list) or as a compiled code object with an arrow pointing to the next instruction to be executed. |

## 1.4 The Inspector Interaction Pane

The interaction pane has two functions: to prompt you and to receive input. If you are not being asked a question, then a read-eval-inspect loop is active. Any forms you type are echoed in the interaction pane and evaluated. The result is not printed, but rather inspected. When you are prompted for input, usually due to having invoked a menu operation, any input you type at the read-eval-inspect loop is saved away and erased from the interaction pane. When the interaction is finished, the input is re-echoed and you can continue to type the form.

## 1.5 Special Characters Recognized by the Inspector

Some special keyboard characters are recognized when not in the middle of typing in a form.

- c-z** Exits and deactivates the Inspector.
- BREAK** Runs a break loop in the typeout window of the bottom-most inspection pane.
- ESCAPE** Reads a form, evaluates it, and prints the result instead of inspecting it. On the LM-2 use the **ALTMODE** key.

## 1.6 The Inspector Menu Pane

The menu pane (to the right of the history pane) displays these infrequently used but useful commands:

- [Exit]** Equivalent to **c-z**. Exits the Inspector and deactivates the frame.
- [Return]** Similar to **[Exit]**, but allows selection of an object to be returned as the value of the call to **inspect**.
- [Modify]** Allows simple editing of objects. Selecting **[Modify]** changes the mouse sensitivity of items on the screen to only include fields that are modifiable. In the typical case of named slots, the names are the mouse-sensitive parts. When the field to modify has been selected, a new value can be specified either by typing a form to be evaluated or by using the mouse to select any normally mouse-sensitive object. The object being modified is redisplayed. Clicking right at any time aborts the modification.
- [DeCache]** Flushes all knowledge about the insides of previously displayed objects and redisplayes the currently displayed objects.
- [Clear]** Clears out the history, the cache, and all the inspection panes.
- [Set] \** Sets the value of the symbol **\** by choosing an object.



## 2. Flavor Examiner

The Flavor Examiner is available via **SELECT X** or the system menu. This is strictly an interim program; it is supported fully in Release 5 but will eventually be incorporated into the Inspector.

Use the **HELP** command to learn how to use this new feature.

The Flavor Examiner utility lets you examine the structure of flavors defined in the Lisp environment. The Flavor Examiner window is divided into six panes.

|                     |                     |
|---------------------|---------------------|
| flavor history pane | method history pane |
| examiner pane       | Edit<br>Lock        |
| examiner pane       | Edit<br>Lock        |
| examiner pane       | Edit<br>Lock        |
| interaction pane    | Clear<br>Help       |

The examiner panes (the three middle panes) list the answer to a query. The edit item of each examiner pane places the contents of the pane into a Zmacs possibilities buffer. The lock item for a examiner pane prevents the pane from being updated.

You enter a flavor name or method-spec into the interaction pane (the bottom pane).

To get started, type the name of a flavor in the interaction pane.

Methods are listed in the following format:

MESSAGE-NAME method-type method-combination-type FLAVOR

If the method-combination-type is **:case**, this format is used:

MESSAGE-NAME SUBMESSAGE-NAME method-type method-combination-type FLAVOR

Clicking on a flavor results in these actions:

- A left click on a flavor presents a menu of flavors and methods related to the flavor. (Note that automatically generated methods to get and set instance variables and methods associated with **si:vanilla-flavor** are not listed.)

- A middle click on a flavor presents a menu of related instance variables.
- A right click on a flavor presents a menu of operations on the flavor, including edit and inspect.
- Any click on a flavor places it in the flavor history pane if it is not already there.

Clicking on a method results in these actions:

- A left click on a method lists the instance variables to which the method refers.
- A middle click on a combined method lists the methods used to build the combined method.
- A middle click on a noncombined method lists all methods for that message from any flavor.
- A right click on a method presents a menu of operations on the method, including [arglist], [documentation], [edit], [inspect], [method spec], [trace], and [disassemble], unless the method is pseudocombined.
- Any click on a method places it in the method history if it is not already there.

Clicking on an instance variable results in these actions:

- A left click on an instance variable lists the methods that refer to the instance variable.
- A middle click on an instance variable shows the default value of the instance variable.

## Index

|          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |          |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| <b>A</b> | <p style="text-align: center;"><b>A</b></p> <ul style="list-style-type: none"> <li>Inspecting an array 3</li> <li>Mouse cursor as an arrow 1</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | <b>A</b> |
| <b>B</b> | <p style="text-align: center;"><b>B</b></p> <ul style="list-style-type: none"> <li>BREAK Inspector command 4</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | <b>B</b> |
| <b>C</b> | <p style="text-align: center;"><b>C</b></p> <ul style="list-style-type: none"> <li>Special                             <ul style="list-style-type: none"> <li>c-Z Inspector command 4</li> <li>Characters Recognized by the Inspector 4</li> <li>[Clear] Inspector menu item 5</li> </ul> </li> <li>Inspecting a closure 3</li> <li>Inspecting a compiled code object 3</li> <li>BREAK Inspector command 4</li> <li>c-Z Inspector command 4</li> <li>ESCAPE Inspector command 4</li> <li>SELECT X command 7</li> <li>Inspector commands 5</li> <li>Inspecting a compiled code object 3</li> <li>Mouse cursor as an arrow 1</li> </ul> | <b>C</b> |
| <b>D</b> | <p style="text-align: center;"><b>D</b></p> <ul style="list-style-type: none"> <li>Inspection Pane                             <ul style="list-style-type: none"> <li>[DeCache] Inspector menu item 1, 5</li> <li>Display 3</li> </ul> </li> </ul>                                                                                                                                                                                                                                                                                                                                                                                    | <b>D</b> |
| <b>E</b> | <p style="text-align: center;"><b>E</b></p> <ul style="list-style-type: none"> <li>Inspecting an entity 3</li> <li>Flavor                             <ul style="list-style-type: none"> <li>ESCAPE Inspector command 4</li> <li>Examiner 7</li> <li>[Exit] Inspector menu item 5</li> </ul> </li> </ul>                                                                                                                                                                                                                                                                                                                              | <b>E</b> |
| <b>F</b> | <p style="text-align: center;"><b>F</b></p> <ul style="list-style-type: none"> <li>Inspecting a stack                             <ul style="list-style-type: none"> <li>inspect Flavor Examiner 7</li> <li>frame 3</li> <li>function 1</li> </ul> </li> </ul>                                                                                                                                                                                                                                                                                                                                                                        | <b>F</b> |
| <b>H</b> | <p style="text-align: center;"><b>H</b></p> <ul style="list-style-type: none"> <li>Inspector history pane 1</li> <li>The Inspector History Pane 1</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | <b>H</b> |



|                                      |                                   |      |
|--------------------------------------|-----------------------------------|------|
|                                      | <b>inspect</b> function           | 1    |
|                                      | Inspecting a closure              | 3    |
|                                      | Inspecting a compiled code object | 3    |
|                                      | Inspecting a list                 | 3    |
|                                      | Inspecting a named structure      | 3    |
|                                      | Inspecting a select method        | 3    |
|                                      | Inspecting a stack frame          | 3    |
|                                      | Inspecting a symbol               | 3    |
|                                      | Inspecting an array               | 3    |
|                                      | Inspecting an entity              | 3    |
|                                      | Inspecting an instance            | 3    |
|                                      | Inspecting objects                | 3    |
| Inspector                            | inspection pane                   | 3    |
| The Inspector                        | Inspection Pane                   | 3    |
|                                      | Inspection Pane Display           | 3    |
| Introduction to the                  | Inspector                         | 1    |
| Special Characters Recognized by the | Inspector                         | 4    |
| The                                  | Inspector                         | 1    |
| Using the mouse in the               | Inspector                         | 1, 3 |
| BREAK                                | Inspector command                 | 4    |
| c-Z                                  | Inspector command                 | 4    |
| ESCAPE                               | Inspector command                 | 4    |
|                                      | Inspector commands                | 5    |
|                                      | Inspector history pane            | 1    |
| The                                  | Inspector History Pane            | 1    |
|                                      | Inspector inspection pane         | 3    |
| The                                  | Inspector Inspection Pane         | 3    |
|                                      | Inspector interaction pane        | 4    |
| The                                  | Inspector Interaction Pane        | 4    |
| [Clear]                              | Inspector menu item               | 5    |
| [DeCache]                            | Inspector menu item               | 1, 5 |
| [Exit]                               | Inspector menu item               | 5    |
| [Modify]                             | Inspector menu item               | 5    |
| [Return]                             | Inspector menu item               | 5    |
| [Set \]                              | Inspector menu item               | 5    |
| The                                  | Inspector Menu Pane               | 5    |
|                                      | Inspector window                  | 1    |
| Inspecting an                        | [Inspect] System menu item        | 1    |
| Inspector                            | instance                          | 3    |
| The Inspector                        | interaction pane                  | 4    |
|                                      | Interaction Pane                  | 4    |
|                                      | Introduction to the Inspector     | 1    |

|              |          |             |   |
|--------------|----------|-------------|---|
|              | <b>L</b> | Line region | 1 |
| Inspecting a |          | list        | 3 |

|  |          |                     |           |      |
|--|----------|---------------------|-----------|------|
|  | <b>M</b> | [Clear] Inspector   | menu item | 5    |
|  |          | [DeCache] Inspector | menu item | 1, 5 |
|  |          | [Exit] Inspector    | menu item | 5    |
|  |          | [Inspect] System    | menu item | 1    |
|  |          | [Modify] Inspector  | menu item | 5    |
|  |          | [Return] Inspector  | menu item | 5    |

[Set \] Inspector menu item 5  
 The Inspector Menu Pane 5  
 Inspecting a select method 3  
 [Modify] Inspector menu item 5  
 Mouse cursor as an arrow 1  
 Using the mouse in the Inspector 1, 3

**N** **N** **N**  
 Inspecting a named structure 3

**O** **O** **O**  
 Inspecting a compiled code object 3  
 Inspecting objects 3

**P** **P** **P**  
 Inspector history pane 1  
 Inspector inspection pane 3  
 Inspector interaction pane 4  
 The Inspector History Pane 1  
 The Inspector Inspection Pane 3  
 The Inspector Interaction Pane 4  
 The Inspector Menu Pane 5  
 Inspection Pane Display 3

**R** **R** **R**  
 Special Characters Recognized by the Inspector 4  
 Line region 1  
 [Return] Inspector menu item 5

**S** **S** **S**  
 Inspecting a select method 3  
 SELECT X command 7  
 [Set \] Inspector menu item 5  
 Special Characters Recognized by the Inspector 4  
 Inspecting a stack frame 3  
 Inspecting a named structure 3  
 Inspecting a symbol 3  
 [Inspect] System menu item 1

**W** **W** **W**  
 Inspector window 1

**X** **X** **X**  
 SELECT X command 7

[Set \ ] Inspector menu item . 5