



odyssey Manual

LAMBDA

LMCS Reference Manual

Mince

Scribble

SYSTEM MAP for Release 2.0

** indicates location of tab divider in binder

These manuals are part of your Lambda documentation, but are not part of a binder.

Intro to Lambda
ZetalISP-Plus Commands

Here are the binders and their contents:



BASICS:

- **LMI Lambda Technical Summary
- **LMI Lambda Field Service Manual
- **NuMachine Installation and User Manual



RELEASE NOTES:

- **Release 2.0 Overview & Notes
- **Release 2.0 Inst & Conversion
- **Editing Lambda Site Files
- **Tape Software & Streams
- **Common LISP Notes



LISP 1: The LISP Machine Manual, Part 1

- **Introduction
 - Primitive Object Types
 - Evaluation
 - Flow of Control
 - Manipulating List Structure
- **Symbols
 - Numbers
 - Arrays
 - Strings
- **Functions
 - Closures
 - Stack Groups
 - Locatives
 - Subprimitives
 - Areas
- **The Compiler
 - Macros
 - The LOOP Iteration Macro
- **Defstruct



LISP 2: The LISP Machine Manual, Part 2

- **Objects, Message Passing, and Flavors
- **The I/O System
 - Naming of Files
 - The Chaosnet
- **Packages
 - Maintaining Large Systems
 - Processes
 - Errors and Debugging
- **How to Read Assembly Language
 - Querying the User
 - Initializations
 - Dates and Times
 - Miscellaneous Useful Functions
- **Indices



LISP 3:

- **Introduction to the Window System
- **The Window System Manual
- **ZMAIL Overview
- **ZMAIL



EDITORS:

- **ZMACS Introductory Manual
- **ZMACS Reference Manual
- **Mince



UNIX 1:

- **NuMachine Release and Update Information
- **NuMachine Operating System
- **UNIX Programmer's Manual, V. 1: Section 1
- **Sections 2-8



UNIX 2: UNIX Programmer's Manual, Vol. 2

- **The UNIX Time-sharing System
- UNIX for Beginners - Second Edition
- A Tutorial Introduction to the UNIX Text Editor
- Advanced Editing On Unix
- An Introduction to the UNIX Shell
- Typing Documents on the UNIX System
- A Guide to Preparing Documents with -ms
- Tbl: A Program to Format Tables
- NROFF/TROFF User's Manual
- A TROFF Tutorial
- **The C Programming Language Reference Manual
- Recent Changes to C
- Lint, A C Program Checker
- Make: A Program for Maintaining Computer Programs
- **UNIX Programming: Second Edition
- A Tutorial Introduction to ADB
- Yacc: Yet Another Compiler-Compiler
- Lex: A Lexical Analyzer Generator
- **A Portable Fortran 77 Compiler
- RATFOR: A Preprocessor for a Rational Fortran
- The M4 Macro Processor
- SED: A Non-Interactive Text Editor
- Awk: A Pattern Scanning and Processing Language (2d. ed.)
- DC: An Interactive Desk Calculator
- BC: An Arbitrary Precision Desk-Calculator Language
- An Introduction to Display Editing with Vi
- **The UNIX I/O System
- On the Security of UNIX
- Password Security: A Case History



HARDWARE 1:

- **NuMachine Technical Summary
- **SDI Monitor User's Manual
- SDI General Description
- **Mouse Manual
- **LMI Printer Software Manual
- **VR-Series Monitor
- Z29 Monitor



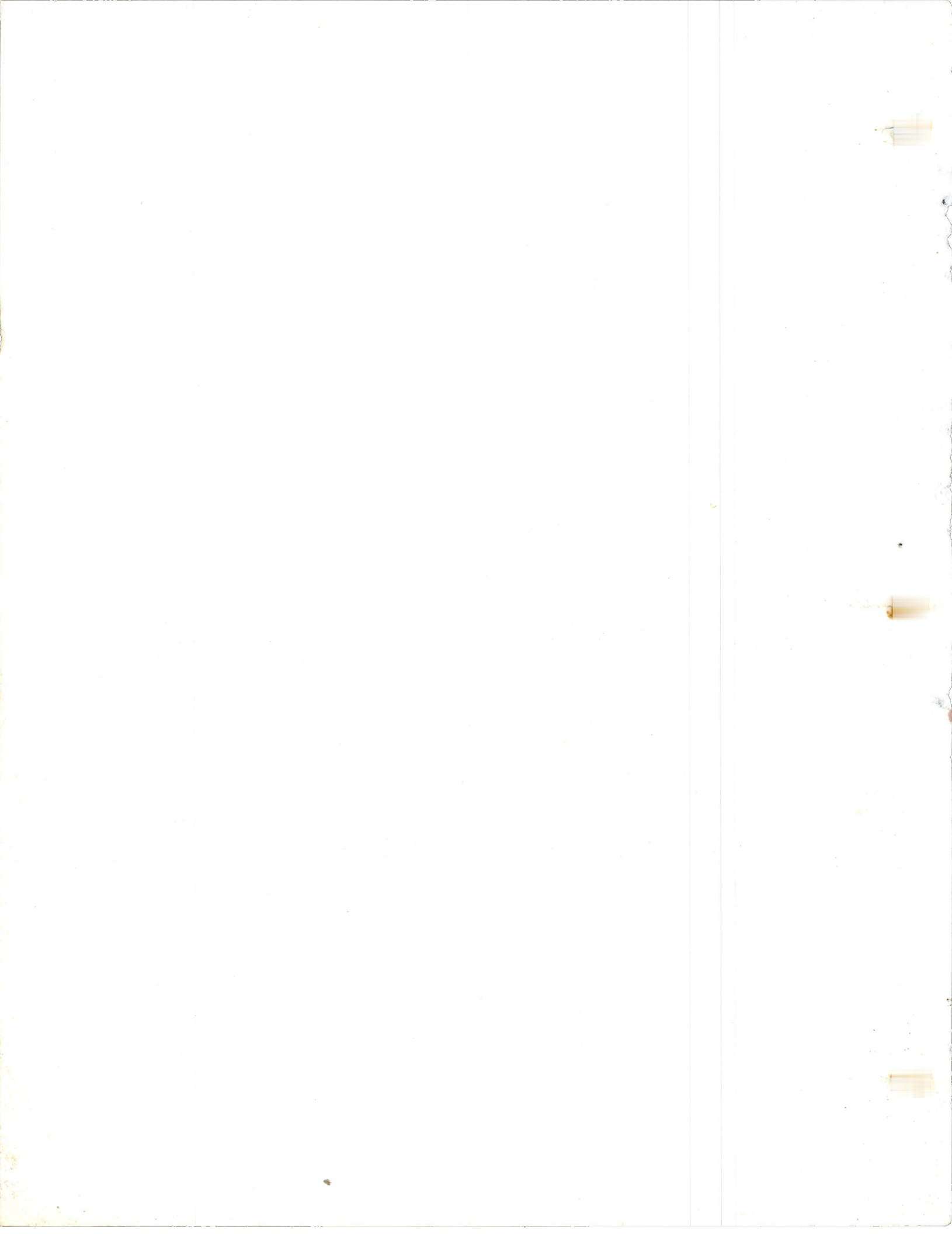
HARDWARE 2:

- **Tape Drive
- **Disk Drive
- **Kermit



OPTIONS:

- ** (varies according to options purchased)
- Prolog
- Interisp
- Fortran: Installation Memo
- Sample
- Libernet Multibus
- Mullin Res Code System
- MTL System



ZMACS Introductory Manual

Sarah Smith

**Published by LMI 6033 W. Century Blvd. Los Angeles CA 90045
USA**

This is a *preliminary* revised version of the *ZMACS Introductory Manual*. Since we are still in the process of checking commands, they are not guaranteed to work. (Input is appreciated.)

However, they are at least as reliable as those in the previous version of this manual. And, we believe, the organization of this manual will make it easier to learn ZMACS.

Please help us to make LMI documentation work better for you! Send comments via your customer dialup mail line to me (username SWRS) or by US Mail to:

Dr. Sarah Smith
Manager, Documentation
LMI
1000 Massachusetts Avenue
Cambridge MA 02139
USA

LMI Lambda[™] is a trademark of LISP Machine Inc.

ZETALISP-PLUS[™] is a trademark of LISP Machine Inc.

Copyright © 1984 LISP Machine Inc.

Table of Contents

Chapter 1 INTRODUCTION	1
1.1 History of ZMACS	1
1.2 Uses of ZMACS: Editing text	1
1.3 Uses of ZMACS: Editing in LISP	2
1.4 Combined functions	2
1.5 ZMACS Files Under UNIX	2
Chapter 2 ZMACS CONCEPTS	3
2.1 Specifications of ZMACS	3
2.2 Two Major Modes: LISP and Text	4
2.3 Integration into ZetaLISP-Plus	4
2.4 Windows, Files, and Buffers	4
2.4.1 Windows	4
2.4.2 Files and Buffers	5
2.4.3 Versions	6
2.5 Deletion, Copying, and Moving	6
2.6 Interface with Code Evaluation and Other System Features	7
2.7 Notes on MINCE (ZMACS under UNIX)	7
Chapter 3 GETTING INTO ZMACS	8
3.1 Login	8
3.2 Entering ZMACS	8
3.2.1 From a LISP Listener	9
3.2.1.1 Changing Modes between Text and LISP	9
3.2.2 By Switching Systems	9
3.2.3 With the Mouse	10
3.2.4 By Listing Buffers	10
Chapter 4 ZMACS TUTORIAL	12
Chapter 5 BASIC ZMACS COMMANDS	13
5.1 Basic Motion Commands	13

Introductory ZMACS

5.1.1 The Character and Word Level	13
5.1.2 The Line and Sentence Level	13
5.1.2.1 Commenting on the Line Level	14
5.1.3 Paragraph and Page Operations	14
5.1.4 Screen operations	14
5.1.5 Buffer Operations	14
Chapter 6 Numeric Arguments	15
Chapter 7 INSERTION AND DELETION	16
7.1 Insertion	16
7.2 Deletion	16
Chapter 8 MARKING, KILLING, AND UNDELETING	18
8.1 Marking	18
8.2 Killing, Moving, and Copying	19
8.3 Yanking	19
Chapter 9 SEARCHING AND REPLACING	21
9.1 Searching	21
9.2 Replacing	22
Chapter 10 CANCELING AND UNDOING	23
10.1 Canceling commands through CONTROL-G	23
10.2 Undoing Drastic Changes	23
Chapter 11 PREFIX COMMANDS AND EXTENDED COMMANDS	25
11.1 Command Completion	25
Chapter 12 LISP COMMANDS	27
12.1 Motion Commands for LISP Code	27
12.2 Killing and Yanking S-Expressions	28
12.3 Parentheses, Marking and Moving S-expressions	29
12.4 Functions (HYPER Key Commands)	30
12.5 Compiling and Evaluating	31
12.6 Regions	32
12.7 Editing Definitions (Meta-Point)	32
12.8 Buffer Commands	33
Chapter 13 THE MOUSE	34
Chapter 14 GETTING HELP	35

14.1 On-line Documentation Commands	36
Chapter 15 INDENTATION AND TABS	38
15.1 Indentation	38
15.2 TAB Indenting	38
15.3 Indentation Commands Involving Regions	39
15.3.1 Changing the TAB Stops	40
15.4 Indenting with Spaces instead of Tabs	40
15.5 LISP Indentation	41
Chapter 16 MORE USEFUL ZMACS FUNCTIONS	42
16.1 Case	42
16.2 Font Changes	42
16.3 Keyboard macros	43
16.3.1 Defining Keyboard Macros	44
16.3.2 Saving Keyboard Macros	45
16.4 Minor Modes	45
16.5 Word Abbreviation	46
16.5.1 Word Abbreviation Commands	46
16.5.2 Creating a Word Abbreviation File	47
16.5.3 Other Useful Word Abbreviation Commands	48
Appendix A BASIC COMMAND SUMMARY	49



Introductory ZMACS

Chapter 1

INTRODUCTION

This manual is designed to introduce ZMACS, the LISP machine's text editor. ZMACS participates fully in the global LISP environment. It can be used for writing and editing English text and for preparing, testing, and debugging LISP programs.

1.1 History of ZMACS

The precursor of ZMACS was EMACS, a powerful text editor developed at the M.I.T. Artificial Intelligence Laboratory for implementation on the DEC PDP-10 computer. EMACS is still widely used and may well already be familiar to the reader. EMACS is written in TECO, an earlier text manipulation language.

EMACS was modified for use on the LISP machine as EINE (a self-referent acronym for "EINE is not EMACS"), which is written in LISP. Further modification of EINE produced ZWEI ("ZWEI was EINE initially"), a text manipulation language in which ZMACS commands are implemented.

1.2 Uses of ZMACS: Editing text

Preparation and editing of English text is a primary application of ZMACS. Because of the importance of this capability, ZMACS incorporates a far larger number and variety of commands than most editors, including a full range of mouse manipulations. The companion ZMACS REFERENCE GUIDE discusses all of these; this guide covers the basics that ZMACS shares with other text editors--the ones you will need first and are most likely to use.

1.3 Uses of ZMACS: Editing in LISP

ZMACS finds a second major application in the LISP environment; because ZMACS is fully integrated into the LISP world, it can be used to prepare and develop LISP programs. Its special capabilities allow it to:

- Monitor the compilation of programs
- Display errors
- Locate the definition of any LISP object.

1.4 Combined functions

Several types of interaction are possible between ZMACS and other LISP Machine capabilities:

- ZMACS can be used in combination with the LISP Listener and the Inspector to create, evaluate, and debug code.
- ZMail, the LISP machine's electronic mail system, creates a ZMACS buffer for the sending and receiving of messages, thus providing full editing capacities for message transfer.
- ZMACS can be extended in LISP code.

1.5 ZMACS Files Under UNIX

While ZMACS itself exists for the LISP Machine, a subset of EMACS is also available on UNIX. MINCE ("MINCE is not complete EMACS") is capable of handling ZMACS files crossloaded from the LISP Processor; it will drive most formatters, including TEX and MINCE's own associated formatter, SCRIBBLE. A dialect of MINCE, configured to be compatible with ZMACS, including Scribble, is available from LMI.

Introductory ZMACS

Chapter 2

ZMACS CONCEPTS

ZMACS handles some concepts in a way that may be different from the text editor(s) you may be used to. Unless you are already familiar with EMACS, read this chapter to become familiar with the concepts behind ZMACS.

2.1 Specifications of ZMACS

ZMACS is a full-screen text editor, able to access any portion of a buffer without the necessity for line searches.

It is by default in "insert mode"; that is, text typed in at a point will insert itself within the file, rather than overwriting previous text. When writing text (rather than LISP code), it will also word wrap.

However, these are only defaults; ZMACS is fully customizable and extensible by the experienced user.

ALL work on the LISP Processor is done in ZMACS; you need to learn no other editor.

It is capable of recognizing the following divisions of text:

- Characters
- Strings
- Words
- Sentences
- Paragraphs
- Regions

- Buffers

In LISP mode, it can also recognize LISP forms.

2.2 Two Major Modes: LISP and Text

Since ZMACS deals with both text and LISP code, ZMACS recognizes two major "modes", one for text, one for LISP. Some ZMACS commands work in only one mode. If a command works in both modes, it works the same way in both.

2.3 Integration into ZetaLISP-Plus

ZMACS is one of the "systems" of ZetaLISP-Plus, like SYSTEM I (the Inspector), SYSTEM Z (Zmail), SYSTEM K (Kermit), and so forth. To change from one of the other ZetaLISP facilities to ZMACS, simply type SYSTEM E (for "Editor"), and you will be connected to a window into the ZMACS system.

2.4 Windows, Files, and Buffers

Conceptually ZMACS is designed to fit into the "systems" environment of ZetaLISP-Plus: That is, into the Window System. Features such as windows and buffers allow ZMACS a robustness and flexibility shared by no other editing system.

2.4.1 Windows

Windows access the facilities of ZetaLISP-Plus, which are grouped together into systems. SYSTEM K, for instance, allows modem-type transfer of data. SYSTEM L, the LISP Listener, evaluates LISP code. SYSTEM I "inspects" and debugs it. A system in operation is called a "process", because it processes data in a specific way. SYSTEM E, the editor, ZMACS, is one of these processes.

Windows get their name because the programmer can look through them at a process. When they are open, data can be thrown through them into a system that processes it. Windows are the access route to systems, and processes take place in windows.

Introductory ZMACS

Physically, they look like rectangles on the screen--"windows" again, into various facilities of ZetaLISP-Plus.

Through ZetaLISP windows, all facilities of the system are visible and reachable. To visualize this, consider all of ZetaLISP-Plus as a factory. Each floor of this factory is devoted to a different part of the production process--evaluation, transfer, editing, and so on. Looking through different windows shows different processes.

Inside this factory, various production "jobs" are being completed. Some jobs are represented by individual files; some are larger, and ZMACS will easily group files into larger job lots such as directories and subdirectories or batches.

Files move through the ZetaLISP-Plus programming "factory", being developed with the help of various systems. They are monitored through the windows associated with each system (and through a system, PEEK, designed specially to monitor). Directions about them are also given through the windows, and material relevant to them enters and leaves.

2.4.2 Files and Buffers

At this point some confusion may be avoided by making some clear distinctions between buffers and files. Files, semi-permanent information stored outside the system environment, are not edited per se. When you edit a file, it is copied into a buffer, a temporary information storage facility within the active system environment.

Each buffer will have the same name as the file from which it was copied. After ZMACS has copied your file into the buffer and you have finished editing the temporary copy, ZMACS uses the edited buffer to write out a new updated version of the file. This process retains the old version of the file unless you delete it. Version numbers, which are incremented with each update, are part of the filename, and distinguish successive versions of your files.

Note -- This applies to LISP only. UNIX does not save version numbers.

Each buffer has its own point, mark and associated information. Only one buffer can be selected at one time. When you select a buffer, ZMACS calls it to the top of the stack on which all existing buffers are kept. Initially the order of the buffers on the stack is the order in which you have created them from the file system. As you use the stack, it will reflect the order of their selection.

The use of buffers has special advantages for LISP programming. This multiple buffer system allows a user to edit several files simultaneously, either by toggling with the CTRL-META-L command or by dividing the screen into multiple windows, to view the contents of several buffers simultaneously. In this way, the user can test LISP programs in the process of being edited without needing to store untested changes in a file. LISP functions can be defined directly out of a ZMACS buffer, allowing the editing and testing of large programs without the need of awkward file system operations.

2.4.3 Versions

ZMACS saves a version of your file every time it is asked to. It does not discard any version unless told.

Versions are discarded in two ways. The current unsaved state of any file is kept in a "buffer", which is a working copy of the file. Any buffer memorializing a bad session state can simply be deleted before it is saved. (The system will query you before doing this.)

Saved versions of files can be deleted by version number. This is called "purging"--a necessary regular household chore in LISP.

2.5 Deletion, Copying, and Moving

In some editors, a deleted item is gone forever. ZMACS is much more forgiving. Any deleted item of any significant size--more than a single letter--is saved in a "kill ring", a special-purpose buffer. There is no limit on the maximum size of a kill; it may be as large as an entire buffer. Any kill can be revived, not necessarily in its original place or its original buffer.

Thus deletion, copying, and moving are the same process in ZMACS. Text can be copied onto the Kill Ring with or without deleting it from its original spot. From the Kill Ring, it can be moved to a new spot or a new buffer. Text can be copied from the Kill Ring any number of times. The Kill Ring can be rotated to reach any item on it.

The Kill Ring serves as a combined short-term storage and moving facility in ZMACS.

Introductory ZMACS

2.6 Interface with Code Evaluation and Other System Features

Since all buffers and files are written in ZMACS, data keyed into ZMACS can be manipulated by any system. It is standard procedure to write LISP code in a ZMACS window, taking advantage of all the editing features, then copy it to a LISP Listener, taking advantage of the SYSTEM L evaluation features.

However, code can also be evaluated and compiled without leaving SYSTEM E by using the commands CTRL-HYPER-C and CTRL-HYPER-E.

2.7 Notes on MINCE (ZMACS under UNIX)

Only a subsection of these facilities work under UNIX. In particular:

- UNIX does not save successive versions; it overwrites. Therefore, only one version of a file is saved under MINCE. (Save other versions, if you need to, by giving them version numbers "by hand" as part of the filename.)
- The Kill Ring in MINCE is only one item long.
- MINCE does not have a special LISP mode. It is preferable to use ZMACS to edit LISP code.

For details, consult the MINCE MANUAL.



Chapter 3

GETTING INTO ZMACS

This chapter will teach you to get into the ZMACS system--and how to get out again. It presumes that your machine has just been warm-booted. If it hasn't been, do so before you start this chapter; you will then be sure of your machine state. Review booting procedures in Chapter 6 of the LMI LAMBDA FIELD SERVICE MANUAL.

3.1 Login

Log onto the machine by typing:

(login "username")

Note that you are in a LISP window and the parentheses enclose a LISP function. Your username is quoted, so that it will not be evaluated. This function will return the value

T

and your name will appear on the mode line at the bottom of the screen.

3.2 Entering ZMACS

You have a choice of four ways to enter a ZMACS buffer:

- From a LISP Listener;
- By switching systems;
- With the mouse;

Introductory ZMACS

- By listing buffers.

3.2.1 From a LISP Listener

From a LISP Listener, type

(ed)

You are now in a ZMACS window, looking at an empty ZMACS buffer. The editor mode line will read

ZMACS (LISP) *Buffer-n*

where n is the identifying number of the buffer.

Buffers are always created in LISP mode. LISP mode is indicated by "(LISP)".

3.2.1.1 Changing Modes between Text and LISP

Since we will work first with text and not with LISP, we will switch major modes and go into another mode, Text Mode.

>>> Type

META-X text <CR>

The notation "(Text)" will replace "(LISP)" on the mode line, indicating that you are now in Text Mode.

Type some text into this buffer:

I Like LISP!

(You can get back into LISP Mode by giving the extended command

Meta-X Lisp Mode

Other major modes are available--for instance, TEX Mode, which is like Text Mode but provides TEX delimiter matching.

3.2.2 By Switching Systems

Go to a LISP window using the System key:

SYSTEM-L

You can recognize the LISP Listener from which you entered the Editor by the function "(ed)", the last item on the screen, which you used to enter the Editor.

>>> Now return to the Editor via

SYSTEM-E

Your original buffer will reappear, with the text "I Like LISP!".

>>> Reenter the LISP window, this time with

CTRL-Z

CTRL-Z takes you back to the window you were in previously, which in this case was a LISP window.

3.2.3 With the Mouse

- Click R2 to call up the SYSTEM MENU.
- Select the SELECT option by pointing at it and clicking L1.
- Select the Editor.

To get a second ZMACS buffer, type

SYSTEM CTRL-E

While the SYSTEM-E command scans the stack for an editor buffer, SYSTEM CTRL-E creates a new editor buffer, whether or not one already exists. Note that this buffer has no text.

You now have two ZMACS buffers, representing two different editing processes. You can check this by listing the buffers you have through one of the following procedures:

- Type in an editor window
Meta-X List Buffers
- Select the List Buffers option on the Editor Menu, as described below.

3.2.4 By Listing Buffers

>>> Call up the EDITOR MENU. Click R1 with the mouse, and the menu will pop up.

Introductory ZMACS

Select the LIST BUFFERS option by pointing at it with the mouse and clicking L1. A list of buffers will appear.

Select the buffer whose number you noted earlier, again by pointing with the mouse and clicking L1. Your text will reappear.

Chapter 4
ZMACS TUTORIAL

ZetaLISP-Plus offers an online ZMACS tutorial of basic ZMACS commands. You can get to the tutorial in either of two ways:

- From the LISP Listener, call the function
 (teach-zmacs)
- From the editor, call the command
 CTRL-META-X Teach Zmacs

That is, simultaneously hold down the CTRL, META and X keys and then type

Teach Zmacs

The ZMACS Tutor introduces ZMACS in an interactive, "hands-on" manner. The commands introduced in the Tutor are reviewed and expanded in the next chapters.



Chapter 5

BASIC ZMACS COMMANDS

For the following exercises, put yourself into a buffer in Text Mode and type some text in it.

5.1 Basic Motion Commands

5.1.1 The Character and Word Level

ZMACS recognizes four directions in which the cursor can move on the screen: Back, Forward, to the Next line, and to the Previous line. (For vertically oriented people, "next" and "previous" are the same as "down" and "up".) These directions are conveniently bound to the four characters B, F, N and P.

Typing any of these four characters with the CTRL key will move the cursor one character in the chosen direction. Typing B or F with the META key will move the cursor one word backward or forward.

>>>Try moving the cursor through your text: forward and backward, upward and down.

5.1.2 The Line and Sentence Level

ZMACS will move the cursor to the beginning and end of lines, like most editors. CTRL-A moves the cursor to the beginning of its current line; CTRL-E, to its end. (If you type to the end of the line in both modes and the line wraps, CTRL-A takes you to the beginning of the wrapped line.)

However, unlike most editors, ZMACS is also capable of recognizing sentence breaks. META-A and META-E will move the cursor to the beginning and end of its current sentence.

5.1.2.1 Commenting on the Line Level

Besides the standard uP and down, CTRL-P and CTRL-N, ZMACS allows you to move up or down and add comments to your text. This is mode-dependent; ZMACS will insert into your buffer whatever the appropriate comment character is, and put your cursor there. To add a comment to your Next or Previous line, type META-N or CTRL-N.

(To add a comment to the current line, type the appropriate comment character.)

Note: If you want to add comments to your code, you should do it in Editor mode. If you do it in a LISP Listener window, the machine will discard your comments as it evaluates your code.

5.1.3 Paragraph and Page Operations

ZMACS requires a blank line between paragraphs. META-[and META-] will move the cursor to the beginning or end of a paragraph--that is, to the last or next blank line.

CTRL-[and CTRL-] will move to the beginning and end of a page.

5.1.4 Screen operations

CTRL-V moves the cursor to the next screenful of text. (Try this with a large file, such as the ZMACS Tutorial.) META-V moves the cursor to the previous screenful of text. Screens are a little less than the amount of text that will fit on a screen; for convenience, there is an overlap of several lines.

In a file with more than one screenful of text, CTRL-L moves the line where the cursor is to the middle of the screen.

5.1.5 Buffer Operations

META-< (left arrowhead or left "bracket") moves the cursor to the beginning of its buffer. META-> moves it to the end of its buffer.

Chapter 6

Numeric Arguments

Motion commands, and many other types of commands, may be given numeric arguments. These are typed before the main command. A numeric argument consists of a modifier key and a number, typed immediately preceding the command to be affected. Any of the four keys, CTRL, META, SUPER, and HYPER, can be used. You may either hold down the key as you type the number, or type CTRL-U, then the number. Then type the command you want. For example, to move forward 35 characters,

>>> Type either

CTRL-3, CTRL-5, CTRL-F

or

CTRL-U 35 CTRL-F

The cursor should move forward 35 characters.

CTRL-U by itself will cause the following command to be executed four times. Successive CTRL-U's have a multiplicative effect; typing CTRL-U CTRL-U before a command will cause the command to be executed 16 times.

Numeric arguments may be given for many commands, and some will take negative as well as positive arguments--for instance, those for moving up and down the stack. Toggle arguments often use a positive numeric argument to equal "yes", a negative numeric argument to equal "no". A negative argument is given by holding down one of the modifier keys (CTRL, META, HYPER, or SUPER), then typing the minus (hyphen) key, and then the number.

Chapter 7

INSERTION AND DELETION

7.1 Insertion

To insert text in standard Text Mode, simply move your cursor to where you want the text and type it in.

You can change to Overwrite Mode, in which text typed in destroys the text "under" it, with the extended command

Meta-X Overwrite

Note that the notation "(overwrite)" will now appear in the mode line. For example, if the cursor is on the "w" in the word "Overwrite" and you key in the word "right", the new word will read "Overrightwrite" in Text Mode. In Overwrite Mode it will read "Overright".

Overwrite Mode is a toggle command. It can be turned off by calling it again:

Meta-X Overwrite

7.2 Deletion

ZMACS deletes text in a group of any size, from one character up. Small deletions are handled by CTRL and META commands. Most of the small deletions can be yanked back with CTRL-Y if you decide you didn't want to delete them after all. Here is a list of deletion commands:

RUBOUT	Deletes the character left of the cursor
CTRL-D	Deletes the character under the cursor
META-RUBOUT	Kills word left of cursor (CTRL-Y yanks it back at cursor position)
META-D	Kills word right of cursor (CTRL-Y yanks)

Introductory ZMACS

CLEAR INPUT	Kills from beginning of line to cursor (CTRL-Y yanks)
CTRL-K	Kills from cursor to end of line (CTRL-Y yanks)
CTRL-X RUBOUT	Kills from beginning of sentence to cursor (CTRL-Y yanks)
META-K	Kills from cursor to end of sentence (CTRL-Y yanks)
CTRL-X K	Kills a buffer. The default is the current buffer, but it gives you a choice.
META-X Delete File	Deletes a file. The default is the file corresponding to the current buffer.

Large deletions are explained in the chapter called "Marking, Killing, and Undeleting".

Chapter 8

MARKING, KILLING, AND UNDELETING

An important concept in ZMACS is the region. A region is any area of text that you delimit by "marking" a beginning and end. Marking regions allows you to kill, move, and copy them.

8.1 Marking

To set one end of the region to be marked, either type CTRL-SPACE or click the middle mouse button once. Move the cursor forward or backward until you reach the other end of the region. The cursor position will mark the other end. The region is now ready to be deleted, copied, or moved.

Some areas may be marked with keystroke commands. For instance:

META-@	Marks to end of word
META-l	Marks entire word cursor is on
META-H	Marks entire paragraph cursor is on
CTRL-X CTRL-P	Marks entire page cursor is on
CTRL-<	Marks to beginning of file (does not move cursor)
CTRL->	Marks to end of file (does not move cursor)

Another way to mark a region is to push the left mouse button at one end of the desired region, and hold it down while you move the mouse to the end of the desired region. When the correct region is marked, release the button.

All marking automatically underlines the marked text.

Clicking L1 (with the mouse) unmarks any marked region and moves the cursor to the arrow position.

Clicking M1 (with the mouse) marks an entire word, no matter which letter the cursor is over.

Introductory ZMACS

8.2 Killing, Moving, and Copying

When a region is marked and killed, it goes onto the "kill ring," a special-purpose ZMACS buffer.

One purpose of the "kill ring" is to temporarily hold recent deletions, so that they can be reconsidered. The second purpose is far more important: to move them. The most usual way of moving text or code in ZetaLISP-Plus is through "killing" it at its original location, moving the point to the desired new location, and resurrecting the killed text at the new location.

Killed items can be any length so long as the length is more than one letter (the kill ring does not save single-letter deletions). The default capacity of the kill ring used to be eight slots; as of System 98 this has been changed, and the kill ring now contains the entire "kill history" of a session.

Items may be copied onto the kill ring without being deleted at their original location.

An item may be copied from the kill ring more than once; this is a useful feature when text is to appear at more than one location.

Here are the basic commands for killing:

CTRL-W	Puts a marked region onto the kill ring.
META-W	Copies a marked region onto the kill ring without erasing it at its original place.

8.3 Yanking

"Yanking" is the converse of killing. It retrieves text from the kill ring.

CTRL-Y	Retrieves text at top of kill ring (most recently killed text)
META-Y	Rotates the kill ring one up (retrieving next-to-most recently killed text). You may call this repeatedly, retrieving each previously

Introductory ZMACS

killed text, until you find the text you want.

Repeating META-Y will cycle through the kill ring. META-Y will take a numeric argument.

Entire buffers can be killed with the command CTRL-X K. The whole buffer then becomes an item on the kill ring.



Chapter 9

SEARCHING AND REPLACING

ZMACS offers the facility of searching for strings. It has two "replace" facilities, allowing the global replacement of all occurrences of a particular string, or querying you case by case.

9.1 Searching

Searches in ZMACS are incremental and immediate; the program searches for the first instance of your string as you enter it. Your string will appear in the echo area.

CTRL-S string Searches forward for string. To move to the next occurrence of string, retype CTRL-S.

CTRL-R string Searches backward for string. To move to the next most recent occurrence of string, retype CTRL-R.

RUBOUT To move backwards by occurrence within your search, hit RUBOUT until you arrive back at the first occurrence of string. At that point RUBOUT deletes characters within string and auto-searches for the shorter string.

ALTMODE Exits from search.

To make changes, exit from search. Then make changes in your text. To search for the next occurrence of string, type CTRL-S twice (or CTRL-R twice, if you are searching backward). The second call to CTRL-S defaults to searching for the string you last searched for.

>>> By using a key word, you can search your text for sections that may require more work or updating of information. This is particularly useful should you find yourself making changes in a text that someone else has written.

9.2 Replacing

CTRL-% string1 <RETURN> string2 <RETURN>
Replaces string1 with string2 throughout

META-% string1 <RETURN> string2 <RETURN>
Replaces some occurrences of string1 with
string2, querying each replacement

Type SPACE for "replace," RUBOUT for "don't replace." Terminate
with ALTMODE.

Pressing the HELP key after calling this command documents it
online.

Chapter 10
CANCELING AND UNDOING

10.1 Canceling commands through CONTROL-G

Any command requiring more than one keystroke can be aborted, at any time before you have pressed the final key, by typing CTRL-G.

10.2 Undoing Drastic Changes

You may find that you have accidentally made a major change in your buffer and you do not know what you have done or how to undo it. The Undo command is an excellent alternative to panic.

META-X Undo, HYPER-CTRL-U

Undoes the last "do-able" command. It works on the last reversible command made. It doesn't matter if you have moved the cursor since the last change.

Before Undo changes your buffer, it will prompt you with the kind of change it plans to undo (i.e., kill, fill, case-convert, etc.) and query you for permission to go ahead. Responding "Y" will implement the Undo.

Remember that Undo will undo the last do-able change, whether or not it is what you intended. You should check the prompt before answering "Y" or you may undo the wrong thing.

When the Undo is completed, you will see a message informing you what has been done and what to do if you don't like it. For instance, if the Undo has been performed on a Yank, you will see: "Yank undone. Hyper-Control-U to undo more, Hyper-Control-R to undo the Undo."

If you use HYPER-CTRL-U repeatedly, Undo will cycle back down the stack of commands, undoing them one by one. This can be

unnerving. META-X Redo changes back each Undo, going back up the stack.

Chapter 11

PREFIX COMMANDS AND EXTENDED COMMANDS

ZMACS recognizes several forms of "prefix" commands: that is, commands using the control keys. Other systems have one prefix key, the CONTROL key, or two, the CONTROL and ESCAPE keys. The Lambda has four, CTRL, META, SUPER and HYPER. The combination of four prefix keys, three mouse keys, and extended commands allows complete customization of the ZMACS environment, without the fear of ever running out of keys.

The CONTROL and META keys are used in combination with other keys to produce simple single-stroke commands. But in combination with X, they produce higher-level commands that can recognize further command keys and even command phrases. CTRL-X commands take one further character (either a CTRL or non-CTRL key); they are called "character extended commands". META-X commands take a further phrase and are called "named extended commands."

Many of the CTRL and META commands prompt for further information such as a buffer name, a yes-no response, or a string. These prompts are designed to be self-explanatory. If they cause any difficulty, type

HELP C Gives online documentation for any command. You must type in the command name at the prompt.

Type any key to remove this documentation and return to your buffer.

11.1 Command Completion

The LISP machine can complete partial text commands that are intelligible to it. After typing in a shortened form of a command that the machine can recognize, you can ask it to complete the command and/or execute it. You do this by typing special keys to end the command. These are:

- SPACE Completes the word you are typing and leaves it displayed in the echo area, pausing to allow you to type in the rest of the command.
- ALTMODE Completes the rest of the command and leaves the line displayed in the echo area, allowing you time to cancel the command. To call the command, type RETURN.
- END If the command is completable, completes it and executes it.
- RETURN Tries to complete the command and executes the result, whether the completion was successful or not. It's best to use END, which won't execute if the command is not completed.

When you call the command and nothing happens, it's most likely that the command name still isn't complete. Type HELP or CTRL-? to see a list of the possible completions of what you have typed.

Another handy command is CTRL-\. This lists alphabetically all commands that contain any of the words in your partial command name. For example, if your incomplete string is "List Buffer", and you type CTRL-\., you'll see a list of all commands containing either the word "list" or the word "buffer". This list is mouse sensitive, so you can select a command with Ll.

Introductory ZMACS

Chapter 12

LISP COMMANDS

LISP forms present problems different from text because of the complexities of their nested subforms; a special set of commands are implemented in the LISP environment to move you accurately through the intricacies of LISP syntax.

These forms will work only when you are in the "LISP Mode" of ZMACS. You can tell you are in this mode when the Mode Line at the bottom of your screen says "ZMACS (LISP)".

You can get into LISP Mode from Text Mode by typing

 META-X lisp

and back to Text Mode by typing

 META-X text

12.1 Motion Commands for LISP Code

CTRL-META-F Moves the cursor one or more s-expressions forward

CTRL-META-B Moves the cursor one or more s-expressions backward

Note the analogy to the text motion commands CTRL-F, META-F, CTRL-B and META-B.

Motion up and down the list structure in LISP does not follow exactly the model for line motion (CTRL-P and CTRL-N). The commands are:

CTRL-META-D Moves the cursor forward and down one or more levels of list structure

CTRL-META-U, CTRL-META-(
Either of these commands moves the cursor up and backward one level of list structure. If called inside of a string, moves back up out of that string.

CTRL-META-), CTRL-META-)
Move the cursor forward and up one level of list structure. If called inside of a string, move up out of that string.

To summarize these last three commands:

CTRL-META-D forward and down
CTRL-META-(backward and up
CTRL-META-) forward and up

(There is no need for "backward and down".)

CTRL-META-A, CTRL-META-[
Move the cursor to the beginning of the current defun (top-level LISP form)

CTRL-META-E, CTRL-META-]
Move the cursor to the end of the current defun (top-level LISP form)

Note the analogy to the text moving commands META-A and META-E.

CTRL-META-R Reposition Window. This command repositions the buffer, trying to display all of the current defun on the screen.

12.2 Killing and Yanking S-Expressions

Killing s-expressions is analogous to killing lines and sentences. The commands are:

CTRL-META-K Kills forward one or more s-expressions

CTRL-META-RUBOUT Kills backward one or more s-expressions

CTRL-Y Yanks the s-expression back from the kill buffer.

12.3 Parentheses, Marking and Moving S-expressions

- META- (Inserts a pair of parentheses around the cursor position with the cursor between them. Then you can write your S-expression without having to remember to add a closing ")". It will already be balanced.
- META-) Moves the cursor past the next close parenthesis, ")", and adjusts the indentation of the following line
- CTRL-META-@ Sets the mark one s-expression from point. Use a numeric argument to mark more than one s-expression.
- CTRL-META-H Puts point and mark around current defun (top-level LISP form).
- CTRL-META-T Interchanges the S-expressions before and after the cursor. With a positive argument, it takes the S-expression to the left of the cursor and moves it "n" S-expressions forward. The reverse is true with a negative argument: it takes the S-expression to the left of the cursor and moves it "n" S-expressions to the right. This command saves you from having to mark S-expressions before moving them. (The "T" in the command stands for "transpose".)

For example, assume you have a series of S-expressions:

a b c (cursor) d e f)

Calling "CTRL-2 CTRL-META-T" will result in:

a b d e c f

Calling "CTRL-minus-3 CTRL-META-T" on that result will give you:

a c b d e f

12.4 Functions (HYPER Key Commands)

This group of commands, using the HYPER key, deal with the definition of LISP functions. Most of these commands also work in the LISP Listener. The commands that apply to regions default to the current defun (top-level LISP form) if no region is specified.

CTRL-HYPER-A Prints the argument list of the function to the left of the cursor.

Note: The function must already have been defined. For example, if the function is a regular piece of LISP code, it is already defined. But, if it's a function defined by a user, it must have been compiled for the LISP machine to print its argument list.

CTRL-HYPER-C Compiles the contents of the current region. If there is no active region, it compiles the top-level S-expression which the cursor is over.

CTRL-HYPER-D Prints out brief documentation for the function the cursor is over. The documentation will be displayed in a temporary window. Type a space to remove the window.

CTRL-HYPER-E Evaluates the current region and returns the value in the Echo area. If there is no current region, it evaluates the top-level S-expression the cursor is over.

CTRL-HYPER-S Moves the cursor to the next occurrence of the pattern specified. A numeric argument repeats the last search *n* times. The pattern specified must be a list. You are prompted for the pattern in the Echo Area.

CTRL-HYPER-V Prints information about the variable (not the function) before the cursor. The information includes:

- Whether or not the variable has been declared special
- Whether it has a value

Introductory ZMACS

- Whether it has documentation put on by DEFVAR. If none of this information is available, CTRL-HYPER-V looks for lookalike symbols in other packages.

META-HYPER-D Prints the arguments and documentation for a function. It prompts you for the function. Typing a RETURN defaults to the function before the cursor. You may also choose a function with the mouse.

META-HYPER-E Evaluates the current region and types out the result in a temporary window. Type a space to remove the window.

CTRL-META-HYPER-E Evaluates the current defun by turning DEFVAR's into SETQ's.

For these commands, the SHIFT key can be used in place of the HYPER key. The HYPER key, however, CANNOT ever be used in place of the SHIFT key (since when you use the SHIFT key in place of the HYPER, it works through the HYPER key). If you are working in ZMACS and want to define a new use for the HYPER key, make sure you use the HYPER and not the SHIFT key.

12.5 Compiling and Evaluating

CTRL-HYPER-C Compiles the contents of the current region. If there is no current region, it compiles the top-level S-expression that the cursor is over.

CTRL-HYPER-E Evaluates the current region in the Echo area. If there is no current region, it evaluates the top-level S-expression the cursor is over.

META-HYPER-E Evaluates the current region and types out the result in a temporary window. Type a space to remove the window.

CTRL-META-HYPER-E Evaluates the current defun by turning DEFVAR's into SETQ's

12.6 Regions

These all work with the current top level expression (defun) if there is no region marked.

- CTRL-EPSILON (Greek E) Compiles the current region
- META-HYPER-E Evaluates the current region and types out the result in the typeout window
- CTRL-HYPER-C Compiles the contents of the current region. If there is no current region, it compiles the top-level S-expression that the cursor is over.
- CTRL-HYPER-E Evaluates the current region in the Echo area. If there is no current region, it evaluates the top-level S-expression the cursor is over.

12.7 Editing Definitions (Meta-Point)

A special type of command, "Meta-Point" or "Meta-Period", allows you to edit function definitions. This type of command finds the code containing a function's definition and puts it in an editor buffer. Then you may edit it.

NOTE: Be careful with these commands. One of the greatest advantages of LISP is its complete programmability; however, you could easily redefine something to your disadvantage.

These commands are invaluable to users who wish to study source code.

- META-. Places the source code containing the definition of the specified function in a buffer. You may then edit that definition or just study it. It prompts you for the function in the Echo Area.
- CTRL-META-. Edits the definition of a key command or extended command. It will prompt you with "Key whose command to edit". Then you should type either a key command or an extended command. Instead of executing that command, the editor will place the

Introductory ZMACS

source code defining that command in an editor buffer. Now you may either edit the definition or just study it.

12.8 Buffer Commands

CTRL-META-X List Buffer Changed Functions

Lists any sections which have been edited in the current buffer.

META-X Sectionize Buffers

Reparses a buffer for definitions. That is, it searches through the buffer for new functions and saves them. Use this command when you have added new functions to the file, so you can then use META-. on them. (See the command META-. in the previous section.)

Chapter 13

THE MOUSE

The mouse greatly facilitates operations in both Text and LISP modes. In most cases, the mouse functions are the same for both modes. One major exception concerns marking regions with the middle mouse button.

In both modes, the mouse button is an extremely efficient tool for marking and manipulating regions. To review, you can mark a region by positioning the cursor at one end of the region you wish to mark. Then click and hold the left mouse button down while moving the cursor to the other end of the region you wish to mark. That region will now be marked by underlining (also referred to as "highlighting"). Typing a CTRL-G will deactivate the region. (See also the chapter "Marking, Killing, and Undeleting".)

The main difference in mouse functions between LISP and Text mode is the middle mouse button function. In Text mode, the middle mouse button recognizes words and in LISP mode it recognizes atoms. (An atom is defined by its matched, bounding parentheses, however long the expression.) In Text mode, clicking M2 marks the word under the cursor; in LISP mode, clicking M2 marks an atom. That is, clicking M2 over the middle of a LISP expression highlights it to the nearest set of matching parentheses. In the case of nested expressions, placing the cursor on one of the parentheses and pressing the middle mouse button will mark the atom bounded by that parenthesis. Thus the middle mouse button is an efficient debugging tool for checking the structure of LISP codes. For example, you can use it to check matching parentheses.

In either LISP or Text mode, pressing the middle mouse button at the beginning or after the end of a line creates a region of the line. The one exception to this is in Text mode. If a line starts with blankspace, you can mark it with M2. If the line starts with text, M2 will mark only the first word.

Chapter 14

GETTING HELP

The HELP key accesses the online-documentation system in ZMACS. This is an invaluable tool for finding information.

HELP in ZMACS is a tree structure, based on a menu of possible kinds of help you can get about the system. When you type the HELP key, a prompt appears asking you to type one of nine keys, each of which selects a particular HELP mode. The ninth key listed is HELP. Selecting this HELP, that is, typing the HELP key a second time, explains the other eight keys which you can select to get help.

HELP prints its online documentation in a temporary window, overlying the main window. To remove the temporary window, type a space.

Here is a list of the first 8 keys under the HELP system.

<HELP> A Apropos. Displays a list of all functions containing the given substring. It also tells how to call each command in the list. It prompts you with "Apropos. (substring)". You must then type in a substring.
This is an invaluable command for finding out about existing commands. For example, type the word "buffer" and you'll see a list of all commands with the word buffer in their name.

Apropos listings can also be generated using the META-X command Apropos.

<HELP> C Displays the documentation for the key you type next. This prompts you with "Document command".
Multi-character commands such as Ctrl-X Ctrl-Z may also be entered.

<HELP> D Displays the documentation for a command by name. It prompts you with "Describe command". You must then type the name of the command at the prompt.

- <HELP> L Returns a history list of the last 60 commands typed at the editor. Use this when you're disoriented and want to see what commands you've typed.
- <HELP> U Undo. Undoes the last undoable command done in the current buffer. The prompt will reflect the type of command done and ask whether to undo it. You must then answer "Y" or "N". It even allows you to undo the undo!
- <HELP> V Variable Apropos. Displays the values of variables whose names contain the substring specified.
- <HELP> W Lists all characters that invoke a given command. It prompts you with "Where is?". You must then type the command name.
- <SPACEBAR> Repeats the previous HELP request. Allows you to return to the same HELP option you just used, instead of exiting HELP, and then starting over by calling HELP and the same option again.

Use the ABORT key to exit from HELP.

If you are in a special mode, you may see other options under HELP. For example, if you call HELP while looking at a directory you'll see the option "M". This option lists the special directory commands.

- <HELP> M This Help command is present only in special modes such as DIREDD. It prints a summary of the commands in the directory editor.

14.1 On-line Documentation Commands

Several other commands give various kinds of on-line documentation.

HYPER-CTRL-D Displays brief documentation in the Echo area for the function before the cursor.

HYPER-META-D Displays all the arguments and online documentation for the function you specify. This information is displayed in a temporary overlay window.

Introductory ZMACS

CTRL-= Prints information about the cursor location. In the Echo area, it lists the cursor's x and y coordinates in octal. The x coordinate is expressed in characters, pixels and columns. If there is a region, it prints the number of lines in it.

CTRL-META-? Is identical to HELP.

Chapter 15

INDENTATION AND TABS

15.1 Indentation

Besides the normal uses of indentation in text, indentation is used in LISP mode to make the code easier to read. Each line of code is indented according to its nesting. ZMACS allows for indentation either with tabs or with spaces. A number of different commands exist which allow you to manipulate tabs, for example, some convert tabs to spaces and vice versa.

15.2 TAB Indenting

The simplest form of indenting is with the TAB key. While its effect varies with the mode in use, in Text mode it indents to the next tab stop. Tabs are set every 8 spaces, but can be altered. In LISP mode, TAB adjusts the indentation of the current line according to the nesting of the code,

TAB	Depending on the mode you are in, either indents the line or adjusts the indentation. In Text mode, TAB indents to the next tab stop. In LISP mode, TAB indents the current line according to its nesting. A numeric argument indents "n" lines at once.
META-TAB	Inserts a TAB character at the cursor point in all modes.
LINE	Inserts a carriage return followed by a TAB in Text mode. In LISP mode, LINE inserts a carriage-return and removes any blankspaces before the next character.
META-^	Deletes the indentation at the front of the current line and the preceding carriage-return

Introductory ZMACS

and replaces them with a single space. No space is inserted when the last character on the previous line is a "(" or when the first character on the line you're moving is a ")".

Giving this command a numeric argument, does the same thing, only to the end of the current line (instead of the beginning). The same exceptions apply.

META-^ undoes the line break whether it was produced manually or by Auto Fill.

META-\
Deletes all spaces and tabs around the cursor. Call this command with your cursor in the indentation at the beginning of a line and the text will move flushleft.

META-M, CTRL-META-M
Positions the cursor at the first nonblank character of the current line. The cursor can be anywhere in the line when you call this command.

15.3 Indentation Commands Involving Regions

ZMACS has special commands to enable you to use regions with indentation commands. These are useful for changing the positioning of blocks of text. You can use regions to indent blocks of text.

CTRL-META-\
Changes the indentation of several lines at once. Gives each line which begins in the region the "usual" indentation by invoking Tab at the beginning of the line. A numeric argument specifies the indentation, and each line is shifted left or right to conform. You must mark a region before calling this command.

CTRL-X TAB, CTRL-X CTRL-I
Moves the entire region a specified number of spaces to the right or left. You must mark the region first. Then you must give the command a numeric argument. A positive argument moves the region n spaces to the left. A negative argument moves the region n spaces to the right.

15.3.1 Changing the TAB Stops

META-X Edit Tab Stops

Allows you to set the tab stops. ZMACS will pop up an overlying window which contains a row of ":"s. Each ":" indicates a tab stop. You may now edit this row by moving around the colons with the usual editing commands. When you return to your text buffer, TAB will space according to the new tab stops (in Text mode).

With the default tab settings of every eight columns, the overlying window of tab stops will look something like this:

```

1
2           :           :           :           :           :           :
3
4      123456789 123456789 123456789 123456789 123456789 1234
5     0          10          20          30          40          50

```

except you will see only the colons. The numbers are added to this example to indicate row and column numbers.

15.4 Indenting with Spaces instead of Tabs

ZMACS normally uses both tabs and spaces to indent lines, and displays tab characters using eight-character tab stops. It is possible to change tabs to spaces and vice versa.

META-X Untabify Converts all tabs in a file to spaces

META-X Tabify META-X Tabify is the opposite of META-X Untabify. It replaces spaces with tabs whenever possible, that is, where there are at least three spaces, in order to protect the breaks between sentences from being turned into tabs.

Introductory ZMACS

15.5 LISP Indentation

As in the case of motion commands, LISP forms present special problems, which indentation commands within the LISP environment are designed to accommodate.

- TAB** In LISP mode, TAB aligns the whole line according to its depth in parentheses, regardless of where in the line the TAB is typed.
- LINE** In LISP mode, LINE performs a carriage-return at the cursor and then indents the new line according to its nesting.
- META-^, CTRL-META-^** This command does the opposite of LINE. It deletes the indentation at the front of the current line and the preceding carriage-return and replaces them with a single space. No space is inserted when the last character on the previous line is a "(" or when the first character on the line you're moving is a ")".
- META-** Deletes the indentation of a line. Use META-\ with the cursor anywhere in the beginning whitespace of the line; it deletes all spaces and tabs around the cursor, moving the text flushleft.
- CTRL-META-Q** Indents the following top-level LISP form, no matter how many lines it is composed of. Use this to indent or re-indent some code that you've been editing.

Chapter 16

MORE USEFUL ZMACS FUNCTIONS

16.1 Case

Place the cursor to the left of the word whose capitalization you wish to change.

META-L Changes the following word to lower case
META-U Changes the following word to upper case
META-C Capitalizes the first letter of the following word

If you use one of these commands while in the middle of a word, it will work forward from the cursor, treating the letters between the cursor and the end of the first word as the first word.

Use negative numeric arguments with these commands to change the case of the previous word.

Use both positive and negative numeric arguments to change the case of more than one word in a direction. For example, to lowercase the last four words before the cursor, type:

META <minus key> 4 L

16.2 Font Changes

ZMACS has commands to alter the font of a word or region. To use different fonts in a file, you must set the fonts. You should also list these fonts in the file's attribute list, the first line of the file. First, you should list the fonts to see what they are. Then you should set the fonts with the META-X SET FONTS command. Then as you type your file, you can change the

Introductory ZMACS

font of individual words or complete sections with the font changing commands.

META-X List Fonts

Lists the defined fonts.

META-X Set Fonts Allows you to set the fonts for a file. Each font is bound to a number. Later, when you want to change a word's font with META-J, you will see the font's number.

META-J

Changes the font of the previous or next word. Fonts are specified by number. If you type META-J with an argument, the argument will be the number of the font which will be applied to the previous word. As with underlining and the META- command, if a font change command already exists in the previous word, the font change will be extended one word to the right. If you use META-J with no argument, it will extend the last font change one word forward. With a negative argument, META-J moves the last font change backwards one word.

CTRL-X J CTRL-X CTRL-J

Changes the font of a region. Its argument is the font number, and with a negative argument removes font changes within or adjacent to the region. You must mark the region before using this command.

CTRL-META-J Sets new default font.

(See the online TEACH ZMACS tutorial for more detail on fonts.)

16.3 Keyboard macros

Keyboard macros are user-defined "abbreviations." When you define a keyboard macro you define a sequence of commands as a new command. For example, to move backwards two lines and search for the first new sentence in that line, the sequence of commands would be:

CTRL-P CTRL-P CTRL-A META-E

If for some reason you had to do this often, you could make your work with ZMACS more efficient by defining this sequence as a keyboard macro. While most ZMACS commands are written in LISP

(see Section 1.1), keyboard macros are written in ZMACS command language. While this makes them less powerful than other ZMACS commands, it makes them easier to write.

16.3.1 Defining Keyboard Macros

In order to define a keyboard macro, you execute the series of commands which comprises the definition. That is, you define a keyboard macro by executing it for the first time. From then on, you can repeat the series of commands by simply executing the macro. The following extended commands allow you to define and manipulate keyboard macros.

CTRL-X (Is the command to begin the definition of a keyboard macro.

CTRL-X) Terminates the definition of a keyboard macro.

CTRL-X E Executes the most recently defined keyboard macro.

META-X View Kbd Macro
Types out the keyboard macro you specify to the prompt.

>>> This exercise will define a keyboard macro for the sequence CTRL-P CTRL-P CTRL-A META-E.

1. Type

CTRL-X (

to begin the macro. The message "Macro level:1" will appear in the mode line.

2. Type the sequence of commands to constitute the macro:

CTRL-P CTRL-P CTRL-A META-E

3. As you type them, these commands will be executed. They will also become part of the macro.

4. Type

CTRL-X)

to end the definition of the macro.

5. To invoke the macro, type

Introductory ZMACS

CTRL-X E

The CTRL-X E command will take numeric arguments, repeating the macro the number of times you specify using CTRL-n CTRL-E.

16.3.2 Saving Keyboard Macros

To save a keyboard macro, you must give it a name. This process will define it as a function. The command is:

META-X Name Last Kbd Macro

which will take the last keyboard macro defined, turn it into a function and give it the name you specify. Thereafter, you can invoke it by the command META-X and the name. For example, if you named your file "search", you would invoke it by calling "META-X search". There is currently a bug in calling the macro with this command, but it does get named. You can still use the macro name for other commands.

META-X Install Macro

Installs the macro onto the key you specify to the prompt. You should name the macro with META-X Name Last Kbd Macro before doing this.

META-X Deinstall Macro

Removes an already installed macro from its specified key

Keyboard macros defined in this way are saved in a library, which, when loaded will automatically redefine the keyboard macro.

16.4 Minor Modes

Minor modes are optional modes which allow you to perform special functions while in them. They are entered usually by calling an extended command made of META-X and their mode name. Calling this same command from within the mode also toggles out of that mode, back to the previous mode. For example, to both enter and leave Overwrite Mode, you would call the command "META-X Overwrite Mode".

A few of the more useful minor modes are:

META-X Overwrite When you type in text in Overwrite mode, the characters you type in replace the existing text rather than pushing it over as in Text mode. To leave Overwrite mode and return to the previous mode, simply retype this command. It toggles you into and out of the mode.

META-X Word Abbrev Mode

This mode is especially useful for text involving any kind of complicated terminology. It allows you to define self-expanding abbreviations. For example, you can define the abbreviation "spit" for the phrase "Systems Programmer in Training". Then as soon as you type "spit" and a space, ZMACS fills out the complete phrase. You may toggle out of this mode by calling this command again.

Use the HELP online documentation with the Apropos option to generate a list of commands with the word "mode" in their name. Type: HELP, "A" for Apropos, and the text string "mode". This will introduce you to the many minor modes.

16.5 Word Abbreviation

This section will describe the commands associated with word abbreviation and take you through the creation and use of a word abbreviation file.

16.5.1 Word Abbreviation Commands

META-X Word Abbrev Mode

This command toggles the Word Abbreviation Mode on and off. When Word Abbrev Mode is on, a notification will appear on the mode line in the same parenthesis that contains Text, Auto Fill and other mode information.

META-X Make Word Abbrev

Defines the abbreviation for a long word or phrase. Prompts you for both the phrase and the abbreviation. Your abbreviation is automatically saved into a word abbreviation file.

CTRL-META-X Save Word Abbrev File

Saves any changes in the word abbreviation file.

Introductory ZMACS

META-X Read Word Abbrev File

This command works like the META-X Load Library command, and loads the word abbreviation file you specify as an argument into the word abbreviation library.

16.5.2 Creating a Word Abbreviation File

>>> We will now create a file of word abbreviations pertinent to ZMACS. The order of operations is:

1. Turn on Word Abbreviation Mode using the command META-X Word Abbrev Mode.
2. Create a new buffer in which to write your word abbreviations. Type in the abbreviations, for example:

```
zx  ZMACS
//  Lisp
wb  word abbreviation
zt  the on-line ZMACS tutorial
```

3. When finish typing in your abbreviations, save this buffer into a file with the command META-X Save Word Abbrev File. Remember the name of this file so you can load it in again.
4. Whenever you want to use the Word Abbrev mode, use the command META-X Read Word Abbrev File to load the particular abbreviation file you wish to use.

Go to the file you want to edit. Make sure you are in Word Abbrev mode. You are now ready to use the word abbreviations. Each time you type an abbreviation and the following space, ZMACS will fill in the complete phrase. For this reason you should choose your abbreviations carefully, since the letters you choose followed by a space will automatically produce the abbreviation, whether you intend it or not.

To define an abbreviation while editing a file, use the command

META-X Make Word Abbrev

It prompts you for the abbreviation and the word or phrase it is to stand for. The default filename for a word abbreviation file is WORDAB.DEFNS. To add the abbreviations you have created during your text editing session, use the command

CTRL-META-X Save Word Abbrev File

which will save the new abbreviations into the current word abbreviation file.

16.5.3 Other Useful Word Abbreviation Commands

META-X List Word Abbrev

Lists the words and abbreviations in the currently-loaded word abbreviation file

META-X List Some Word Abbrevs

Give this command an abbreviation as a string and it will list the expansion. Given the expansion, it will list the abbreviation.

Use the HELP online documentation and Apropos option to generate a list of commands containing the substring "Abbrev". Type: HELP, "A" for Apropos, and the text string "abbrev". This will introduce you to other word abbreviation commands.

Appendix A

BASIC COMMAND SUMMARY

BY KEY:

<u><key></u>	<u>CONTROL-<key></u>	<u>META-<key></u>
A	Beginning of line	Beginning of sentence
B	Back char	Back word
D	Delete next char	Delete next word
E	End of line	End of sentence
F	Forward char	Forward word
G	Quit, break	Fill Region
K	Kill to line end	Kill to sentence end
L	Redisplay	Lowercase word
N	Next line	Down comment line
P	Previous line	Up comment line
Q	Quoted insert	Reform paragraph
R	Reverse search	Move to screen edge
S	Search forward	Center line
T	Transpose character	Transpose words
V	Forward screenful	Back screenful
W	Kill marked region	Copy marked region
X	Char. eXtended command	Named eXtended command
Y	Yank most recent kill	Yank previous kill
Rubout	Delete previous char	Delete previous word
<	Mark to file beginning	Move to file beginning
>	Mark to file end	Move to file end

CTRL-X Commands:

CTRL-X CTRL-F	Finds file
CTRL-X CTRL-S	Saves file
CTRL-X F	Sets fill column

META-X Commands:

META-X Replace String	Queries before replacing
META-X Auto Fill Mode	Performs auto fill
META-X Apropos	Lists functions containing a

Introductory ZMACS

CTRL-X RUBOUT Kills from beginning of sentence to cursor (CTRL-Y yanks)
META-K Kills from cursor to end of sentence (CTRL-Y yanks)

Paragraph operations

META-[Moves to beginning of paragraph
META-] Moves to end of paragraph
META-H Marks current paragraph
META-Q Fills current paragraph
CTRL-n CTRL-X F Sets the fill column to n characters

Screen operations

CTRL-V Moves to next screen
META-V Moves to previous screen
CTRL-L Moves cursor position to screen center
CTRL-n CTRL-L Moves cursor position to n lines from top of screen

Page operations

CTRL-Q CLEAR SCREEN Inserts a PAGE character
META CLEAR-SCREEN The same

Buffer operations

CTRL-X B Selects another buffer; default is previous buffer
CTRL-X CTRL-B Displays a mouse-sensitive menu of available buffers
CTRL-X K Kills a buffer; default is current one
CTRL-META-L Moves to most recently displayed buffer. (Good for switching between two buffers.)
CTRL-X CTRL-W Saves the current buffer but allows you to specify a filename that may differ from the default

File operations

CTRL-X CTRL-F Creates or finds a file
META-< Moves cursor to beginning of file
CTRL-< Marks to beginning of file (does not move cursor)
META-> Moves cursor to end of file
CTRL-> Marks to end of file (does not move cursor)
CTRL-X CTRL-S Saves the current file
META-X Delete File Deletes a file. The default is the file corresponding to the current buffer

SUMMARY OF MOTION COMMANDS

	forward	backward	beginning	end
Character level	CTRL-F	CTRL-B		
Word level	META-F	META-B		
Line level	CTRL-N	CTRL-P	CTRL-A	CTRL-E
Sentence			META-A	META-E
Paragraph	META-]	META-[META-[META-]
Page	CTRL-X]	CTRL-X [CTRL-X [CTRL-X]
Screen	CTRL-V	META-V		
Buffer	META->	META-<	META-<	META->

Try the command META-X Generate Wallchart for listing various types of commands. It will list several options. Select the option "all" to generate a list of all keyboard commands.

ZMACS Reference Manual

1915-0000

First Edition, System Version 99

June 1985

Richard Stallman

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

Summary Table of Contents

Introduction	3
1. The ZMACS Frame	5
2. Characters, Commands and Functions	11
3. Invoking ZMACS	15
4. Basic Editing Commands	19
5. Basic Mouse Commands	23
6. Numeric Arguments	27
7. The Minibuffer	29
8. Extended (Meta-X) Commands	35
9. Help	37
10. The Mark and the Region	39
11. Killing and Moving Text	45
12. Undoing Changes	53
13. The Various Quantities Command	57
14. Controlling the Display	59
15. Searching	61
16. Commands for Fixing Typos	71
17. File Handling	75
18. Dired, the Directory Editor	91
19. Using Multiple Buffers	99
20. Multiple Windows	109
21. Major Modes	113
22. Indentation	115
23. Case and Fonts	119
24. Commands for Natural Languages	125
25. Editing LISP Code	135
26. Running and Testing LISP Programs	153
27. Ztop Mode	171
28. Word Abbrevs	175
29. Miscellaneous Commands	179
30. Attributes in Files and in Buffers	187
31. Customization	191
32. Correcting Mistakes and ZMACS Problems	205
Glossary	209
Command Function Index	223
Command Character Index	229
LISP Function Index	233
Variable Index	235
Concept Index	237



Table of Contents

Preface	1
Introduction	3
1. The ZMACS Frame	5
1.1 Point	5
1.2 Typeout	5
1.3 The Echo Area	6
1.4 The Mode Line	7
1.5 Cursor Position Information	8
2. Characters, Commands and Functions	11
2.1 LISP Machine Character Set	11
2.2 Multicharacter Commands	11
2.3 Commands, Functions, and Variables	12
2.4 ZWEI	13
3. Invoking ZMACS	15
3.1 Exiting ZMACS	16
4. Basic Editing Commands	19
4.1 Inserting Text	19
4.2 Changing the Location of Point	20
4.3 Erasing Text	21
4.4 Files	21
4.5 Help	22
4.6 Blank Lines	22
5. Basic Mouse Commands	23
5.1 Cursor Motion and Regions	23
5.2 Killing and Yanking	24
5.3 Scrolling	24
5.4 Mouse-Sensitive Typeout	24
5.5 The Editor Menu	25
5.6 Arguments	25
6. Numeric Arguments	27
7. The Minibuffer	29
7.1 Completion	30
7.1.1 Matching for Completion	32
7.2 Repeating Minibuffer Commands	33
8. Extended (Meta-X) Commands	35

9. Help	37
10. The Mark and the Region	39
10.1 Operating on the Region	40
10.2 Commands to Mark Textual Objects	41
10.3 The Point Pdl	41
10.4 Named Marks	42
11. Killing and Moving Text	45
11.1 Deletion and Killing	45
11.1.1 Deletion	45
11.1.2 Killing by Lines	46
11.1.3 Other Kill Commands	47
11.2 Yanking	48
11.2.1 Appending Kills	48
11.2.2 Yanking Earlier Kills	49
11.3 Other Ways of Copying Text	50
11.3.1 Accumulating Text	50
11.3.2 Copying Text Using Registers	51
12. Undoing Changes	53
13. The Various Quantities Command	57
14. Controlling the Display	59
15. Searching	61
15.1 Searching and Case	63
15.2 Nonincremental Search	63
15.3 Word Search	64
15.4 LISP Pattern Search	65
15.5 Replacement Commands	65
15.5.1 Replace String	65
15.5.2 Query Replace	66
15.6 Other Search-and-Loop Commands	68
15.7 Extended Search Characters	68
16. Commands for Fixing Typos	71
16.1 Killing Your Mistakes	71
16.2 Transposing Text	72
16.3 Case Conversion	72
16.4 Checking and Correcting Spelling	73
17. File Handling	75
17.1 File Names	75
17.2 Visiting Files	76

17.2.1 Visiting Multiple Files	78
17.3 Saving Files	79
17.3.1 Protection against Simultaneous Editing	80
17.4 Reverting a Buffer	81
17.5 Listing a File Directory	82
17.6 Deleting and Expunging Files	82
17.6.1 Expunging and Undeletion	83
17.6.2 Deleting Old Versions	84
17.7 Comparing Files	84
17.7.1 Merging Files	86
17.8 Miscellaneous File Operations	87
18. Dired, the Directory Editor	91
18.1 Deleting Files with Dired	91
18.2 Dired Cursor Motion	92
18.3 Displaying Other Directories	93
18.4 Operations on File Properties	93
18.5 Other File Operations in Dired	94
18.6 Immediate File Operations in Dired	95
18.7 Sorting the Dired Buffer	96
18.8 Balancing Directories Interactively	96
19. Using Multiple Buffers	99
19.1 Creating and Selecting Buffers	99
19.2 Specifying a Buffer with a File Name	101
19.3 Miscellaneous Buffer Operations	102
19.4 Killing Buffers	102
19.5 Operating on Several Buffers	103
19.6 Buffer Groups	104
19.6.1 Creating and Selecting Buffer Groups	104
19.6.2 Searching a Group of Buffers	106
19.6.3 Stepping Through a Buffer Group	107
19.6.4 Buffer Groups and Sectionization	107
20. Multiple Windows	109
21. Major Modes	113
22. Indentation	115
22.1 Tab Stops	117
22.2 Other Styles of Indenting a Line	117
23. Case and Fonts	119
23.1 Case Conversion Commands	119
23.2 Fonts	120

23.2.1 Specifying the List of Fonts	120
23.2.2 Specifying a Font from the List	121
23.2.3 Font-Change Commands	122
23.2.4 Fonts and Copying Text	123
24. Commands for Natural Languages	125
24.1 Text Mode	125
24.2 Words	126
24.3 Sentences	127
24.4 Paragraphs	128
24.5 Pages	129
24.6 Filling Text	130
24.7 Editing Text Formatter Directives	133
24.7.1 Font Change Commands	133
25. Editing LISP Code	135
25.1 LISP Mode	135
25.2 S-expressions and Lists	136
25.3 Defuns	138
25.4 LISP Indentation	139
25.4.1 Customizing LISP Indentation	140
25.5 Automatic Display Of Matching Parentheses	143
25.6 Manipulating Comments	143
25.6.1 Multiple Lines of Comments	144
25.6.2 Commenting Out Code	145
25.6.3 Double and Triple Semicolons in LISP	146
25.6.4 Options Controlling Comments	146
25.7 Case Conventions for LISP Code	147
25.8 Editing Commands Based on LISP Semantics	148
25.9 Editing Without Unbalanced Parentheses	149
25.10 Documentation Commands for LISP Code	150
26. Running and Testing LISP Programs	153
26.1 Sectionization	153
26.2 Compiling LISP Files	157
26.3 LISP Compilation and Evaluation	157
26.4 Compiler Warnings	160
26.5 LISP Debugging Aids in ZMACS	161
26.6 Exploring the Flavor Hierarchy	164
26.7 Possibilities Lists	164
26.8 The Patch Facility	165
26.8.1 Making Patches	166
26.8.2 Private Patches	169

27. Ztop Mode	171
28. Word Abbrevs	175
28.1 Defining Abbrevs	175
28.2 Expanding Abbrevs	176
28.3 Examining and Altering Abbrevs	177
28.4 Saving Abbrevs	178
29. Miscellaneous Commands	179
29.1 Recursive Edits	179
29.2 Sending Mail	179
29.3 Hardcopy Output	180
29.4 Sorting	181
29.4.1 Evaluating Expressions Interactively	182
29.5 Editing Assembly-Language Programs	183
29.6 Major Modes for Other Languages	184
29.7 Dissociated Press	185
30. Attributes in Files and in Buffers	187
30.1 Commands Setting Buffer Attributes	188
31. Customization	191
31.1 Minor Modes	191
31.2 Variables	192
31.3 Keyboard Macros	195
31.3.1 Basic Use	196
31.3.2 Naming and Installing Keyboard Macros	197
31.3.3 Nesting Macro Definitions	198
31.3.4 Executing Macros with Variations	198
31.4 Command Functions and Command Tables	199
31.4.1 Command Functions	199
31.4.2 Changing Comtabs from ZMACS	199
31.4.3 Meta-X Availability	201
31.5 The Syntax Table	202
32. Correcting Mistakes and ZMACS Problems	205
32.1 Quitting and Aborting	205
32.2 Dealing with ZMACS Trouble	206
32.2.1 Subsystems and Recursive Editing Levels	206
32.2.2 Garbage on the Screen	206
32.2.3 Garbage in the Text	207
32.2.4 ZMACS Hung and Not Responding	207
Glossary	209

Command Function Index 223

Command Character Index 229

LISP Function Index 233

Variable Index 235

Concept Index 237

Preface

This manual documents the use and simple customization of the display ZMACS editor. The reader is not expected to be a programmer. Even simple customizations do not require programming skill, but the user who is not interested in customizing can ignore the scattered customization hints.

This is primarily a reference manual, but can also be used as a primer. I recommend that the newcomer first use the on-line tutorial TEACH-ZMACS. With it, you learn ZMACS by using ZMACS on a specially designed file that describes commands, tells you when to try them, and then explains the results you see. This gives a more vivid introduction than a printed manual.

On first reading, you need not make any attempt to memorize chapters one and two, which describe the notational conventions of the manual and the general appearance of the ZMACS display screen. It is enough to be aware of what questions are answered in these chapters, so you can refer back when you later become interested in the answers. After reading chapter three you should practice the commands there. The next few chapters describe fundamental techniques and concepts that are referred to again and again. It is best to understand them thoroughly, experimenting with them if necessary.

To find the documentation on a particular command, look in the index if you know what the command is. Characters and command functions have separate indexes just for them.

ZMACS is a member of the Emacs editor family. There are many Emacs editors, all sharing common principles of organization. For information on the underlying philosophy of Emacs and the lessons learned from its development, write for a copy of AI memo 519a, "Emacs, the Extensible, Customizable Self-Documenting Display Editor", to

Publications Department
Artificial Intelligence Lab
545 Tech Square
Cambridge, MA 02139



Introduction

You are about to read about ZMACS, the LISP Machine version of the advanced, self-documenting, customizable, extensible real-time display editor Emacs.

We say that ZMACS is a *display* editor because normally the text being edited is visible on the screen and is updated automatically as you type your commands. See chapter 1 [Screen], page 5.

We call it a *real-time* editor because the display is updated very frequently, usually after each character or pair of characters you type. This minimizes the amount of information you must keep in your head as you edit. See chapter 4 [Basic Editing], page 19.

We call ZMACS *advanced* because it provides facilities that go beyond simple insertion and deletion: filling of text; automatic indentation of programs; viewing two or more files at once; and dealing in terms of characters, words, lines, sentences, paragraphs, and pages, as well as expressions and comments in several different programming languages. It is much easier to type one command meaning "go to the end of the paragraph" than to find that spot with simple cursor keys or the mouse.

Self-documenting means that at any time you can type a special character, the **HELP** key, to find out what your options are. You can also use it to find out what any command does, or to find all the commands that pertain to a topic. See chapter 9 [Help], page 37.

Customizable means that you can change the definitions of ZMACS commands in little ways. For example, if you use a programming language in which comments start with '`<*`' and end with '`*>`', you can tell the ZMACS comment manipulation commands to use those strings. Another sort of customization is rearrangement of the command set. For example, if you prefer the four basic cursor motion commands (up, down, left and right) on keys in a diamond pattern on the keyboard, you can have it. See chapter 31 [Customization], page 191.

Extensible means that you can go beyond simple customization and write entirely new commands, programs in the LISP language. ZMACS is entirely written in LISP and can be replaced, function by function, by each user during a session. Of course, this is true of the entire LISP Machine system, so it may not seem worth mentioning for ZMACS in particular. However, extensible Emacs editors existed before the LISP Machine system. One can just as well say that the LISP Machine is the first system that is entirely as extensible as an Emacs editor.



1. The ZMACS Frame

ZMACS uses a special type of window called a *ZMACS frame*. You can create a new ZMACS frame in several ways, such as by typing **(SYSTEM) Control-E**, and each ZMACS frame is an independent ZMACS editor.

The ZMACS frame is divided into several windows, each of which contains its own sorts of information. The biggest window is the one in which you usually see the text you are editing—the text of the selected ZMACS buffer (see chapter 19 [Buffers], page 99).

1.1 Point

One position in the selected buffer is identified as *point*. This is where most ZMACS editing commands act. Other commands move point through the text, so that you can edit at different places in it.

A blinking rectangular block cursor in the text window shows you the location of point. While the cursor appears to point *at* a character, point should be thought of as *between* two characters; it points *before* the character that the cursor appears on top of. Sometimes people speak of “the cursor” when they mean “point”, or speak of commands that move point as “cursor motion” commands.

The width of the cursor changes to match the character that it is on top of. If that character is a tab character, the cursor is normally the width of a space. However, if the variable `zwei:*tab-blinker-flag*` is set to `nil`, the cursor when before a tab character covers all the horizontal space that the tab character occupies. See section 31.2 [Variables], page 192.

1.2 Typeout

The text window can also display *typeout*. This is output from an editing command that is not actually part of the text being edited; for example, the list of all ZMACS buffers produced by **C-X C-B (List Buffers)**. Typeout can be recognized because it appears sequentially starting at the top of the screen, with a cursor visible at the end, and a horizontal line stretching across the window just below the bottom line of typeout.

The typeout appears there for your information, but it is not part of the file you are editing,

and as soon as you type another command the typeout disappears and the text you are editing comes back. If you want to make typeout go away immediately but not do anything else, you can type a **(SPACE)**. (Usually the command **(SPACE)** inserts a space character, but when there is typeout on the window **(SPACE)** does nothing but get rid of the typeout.) There is always a cursor at the end of the typeout, but this does not mean that point has moved. The cursor moves back to the location of point after the typeout goes away. This is an example of a *typeout window*. These are described in a chapter in the *Window System Manual*.

ZMACS allows you to divert typeout from one command to be inserted in the current buffer instead, with the command **M-X Execute Command Into Buffer**. After giving this command (and typing **(RETURN)** to terminate the command name, of course), type any ZMACS command, possibly a multi-character command starting with **C-X**, **M-X** or **C-Shift-X**. The second command executes normally except for inserting any typeout it would have done. Point is left after the inserted text, and the mark is left before it (but not activated). See chapter 10 [Mark], page 39.

Typeout is often *mouse-sensitive*: it does something if you click the mouse on it. See section 5.4 [Mouse Typeout], page 24.

1.3 The Echo Area

A few lines at the bottom of the screen compose what is called the *echo area*. It is used to display small amounts of text for several purposes.

Echoing means printing out the commands that you type. ZMACS does not echo single-character commands, and usually does not echo short commands, but if you pause for more than a second in the middle of a multi-character command, then all the characters typed so far are echoed. This is intended to *prompt* you for the rest of the command. Once the beginning of a command has been echoed, all the rest is echoed as soon as it is typed; so either the entire command or none of it is echoed. This behavior is designed to give confident users fast response, while giving hesitant users maximum feedback.

If a command cannot be executed, it may print an *error message* in the echo area. Error messages are accompanied by a beep or by flashing the screen. Also, any input you have typed ahead is thrown away when an error happens.

Some commands print informative messages in the echo area. These messages look much like error messages, but they are not announced with a beep and do not throw away input. Sometimes the message tells you what the command has done, when it is not obvious from looking at the

text being edited. Sometimes the sole purpose of a command is to print a message giving you specific information. For example, the command `C-X =` is used to print a message describing the coordinates of point in the window and its line number in the text.

The echo area is also used to display the *minibuffer*, a window three lines tall that is used for reading arguments to commands, such as the name of a file to be edited. When the minibuffer is in use, its rectangular outline appears within the echo area. You can always get out of the minibuffer by typing `(ABORT)`. See chapter 7 [Minibuffer], page 29.

1.4 The Mode Line

The line above the echo area is known as the *mode line*. Usually it starts with `'ZMACS (something)'`; *something* is usually `'Lisp'`. When the mode line looks that way, you are at *top level*, typing ZMACS commands to edit the current buffer. If the mode line does not look that way, you are inside a minibuffer or a recursive edit.

If the mode line starts and ends with brackets, `'[...]'`, then you are in a recursive edit inside of another command. The text in the mode line says what command. See section 29.1 [Recursive Edit], page 179.

While you are supplying an argument in the minibuffer, the mode line contains a *prompt* which explains what sort of text you should enter with the minibuffer, and possibly some information on how to do so (such as, that completion is available, or that extended search characters may be used, or that only `(END)` can be used to terminate the argument). See chapter 7 [Minibuffer], page 29.

At top level, the mode line serves to indicate what buffer and file is being displayed in the selected window; what major and minor modes are in use; and whether the buffer's text has been changed. The top level mode line has this format:

`ZMACS (major minor) * pos bfr (vrs) Macro-level: n`

major 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 (the least specialized), Text mode, LISP mode, C mode, and others. See chapter 21 [Major Modes], page 113, for details of how the modes differ and how to select one.

minor is a list of some of the *minor modes* that are turned on at the moment. 'Fill' means that Auto Fill mode is on. 'Atom' means that Atom Word mode is on. 'Abbrev' means that Word Abbrev mode is on. 'Ovrrt' means that Overwrite mode is on. 'Electric Shift Lock' and 'Electric Font Lock' indicate the minor modes of the same names. See section 31.1 [Minor Modes], page 191, for more information.

The star in the mode line means that there are changes in the buffer that have not been saved on the file server. If the current buffer is not "modified", no star appears.

pos gives an indication of which part of the text is being displayed. An upward arrow is present if there is text in the buffer before what appears at the top of the window. A downward arrow appears if there is more text below the bottom of the window. Both arrows appear if there is more text above and more text below.

bfr is the name of the currently selected *buffer*. Each buffer has its own name and can hold a file being edited; this is how ZMACS can hold several files at once. But 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 make it easy to switch around between several files, and then it is very useful that the mode line tells you which one you are editing at any time. See chapter 19 [Buffers], page 99.

'**Macro-level:** *n*' appears only during the definition of a keyboard macro. *n* is the depth in macro definitions, normally 1. See section 31.3 [Keyboard Macros], page 195.

The mode line is not the same thing as the *wholine*; the *wholine* is a feature of the LISP Machine system in general and appears at the very bottom of the screen no matter what program is running. The ZMACS mode line is a feature specifically of ZMACS, and appears inside the ZMACS frame, above the echo area.

1.5 Cursor Position Information

If you are accustomed to other display editors, you may be surprised that ZMACS does not always display the page number or line number of point in the mode line. This is because the text is stored in a way that makes it difficult to compute this information. Displaying them all the time would be too slow to be borne. They are not needed very often in ZMACS anyway, but there are commands to print them.

C-X - Print row and column of point, following character code, and line number.

C=- Similar but omit line number for greater speed.

The command **C-X = (Where Am I)** can be used to find out the column that the cursor is in, and other miscellaneous information about point. It prints a line in the echo area that looks like this:

```
X=[13 chars|104 pixels|13 columns] Y=53 Char=#o215 Line=2283(29%)
```

The 'X' value says how far the cursor is from the left margin, in three different ways: the number of actual characters, the number of pixels, and the number of columns (which is the number of pixels, divided by the width of a space in font A). The number of characters is often the same as the number of columns, but they can differ when variable-width fonts or multiple fonts are used.

The 'Y' value is straightforward: it is the number of lines *in the editor window* above the cursor. The 'Char' value is the octal code for the character that follows point. (The characters '#o' are to remind you that the character code is in octal.) The 'Line' value is the number of lines *in the buffer* above the line point is on, and the percentage following expresses it as a fraction of the total number of lines in the buffer.

If there is a region, **C-X =** prints additional text saying how many lines are in the region.

The command **C=- (Fast Where Am I)** is like **C-X =** except that it omits the line count. Counting the lines can make **C-X =** slow if the text is not all in core. **C=-** is always fast.



2. Characters, Commands and Functions

In this chapter we introduce the terminology and concepts used to talk about ZMACS commands.

2.1 LISP Machine Character Set

The LISP Machine uses an 8-bit character set, which offers 256 different character codes. Some of these codes are assigned graphic symbols such as 'a' and '='; some have names, such as **RETURN** and **BREAK**; some are completely unassigned.

Any of the 256 possible character codes can appear in files and in ZMACS buffers. Graphic characters in ZMACS buffers are displayed according to the current font. Nongraphic characters that have names are displayed as a lozenge containing the name, except for **RETURN**, which causes a new line to start in the display, and **TAB**, which produces a variable amount of whitespace. **RETURN** characters in the text are informally called *newlines*. Unassigned codes display as a lozenge containing the octal character code.

Keyboard command for ZMACS come from the same 256-character set (though the unassigned codes cannot be typed on the keyboard), but each character can be modified by the four keys **CONTROL**, **META**, **SUPER** and **HYPER**. These are commonly abbreviated as 'C-', 'M-', 'S-' and 'H-' in the names of keyboard characters; thus, **C-M-F** stands for **Control-Meta-F**, an **F** typed with **CONTROL** and **META** held down. Sixteen possible combinations of those keys, times 256 basic character codes, make 4096 possible ZMACS commands. Most of these are not defined!

Alphabetic case makes a difference in ZMACS commands; **C-Shift-P** is not the same character as **C-P**, and the two characters have very different meanings. However, the **CAPS LOCK** key is ignored whenever **CONTROL**, **META**, **SUPER** or **HYPER** is held down.

2.2 Multicharacter Commands

Most of the characters on the keyboard are complete ZMACS commands in themselves, though they may read arguments or ask questions when executed. The characters **Control-X**, **Meta-X**, and **C-Shift-X** are different; any of them serves as just the first character of a multicharacter command.

The character **C-X** is a *prefix character*, which means that it and next character typed combine to make a two-character command. The various two-character combinations starting with **Control-X** are defined independently and they are documented individually in this manual.

Meta-X introduces an *extended command*, which has a long name made of one or more English words. Following **M-X**, you must type the command name, possibly using completion to abbreviate it. See chapter 8 [Extended commands], page 35.

C-Shift-X introduces a three-character command in which the second character specifies a kind of operation (move forward, kill, convert to lower case, etc.) and the third character specifies how large a textual unit to operate on (lines, words, characters, sentences, etc.) See chapter 13 [Various Quantities], page 57.

2.3 Commands, Functions, and Variables

Most of the ZMACS commands documented herein are single characters or two-character sequences. But ZMACS does not directly assign meanings to characters. Instead, ZMACS assigns meanings to *command functions*, and then gives characters their meanings by *connecting* them to command functions. A command function has a *command name*, a name such as **Down Real Line** containing capitalized words separated by spaces. It also has a *definition* which is a LISP program; this is how the command function does what it does. The connections or *bindings* between command characters and command functions are recorded in various *command tables* (or *comtabs*). See section 31.4 [Comtabs], page 199.

When we say that “**C-N** moves down vertically one line” we are glossing over a distinction that is irrelevant in ordinary use but is vital in understanding how to customize ZMACS. It is the command function **Down Real Line** that is programmed to move forward by characters. **C-N** has this effect *because* it is connected to that command function. If you reconnect **C-N** to the command function **Forward Word** then **C-N** will move forward by words instead. Reconnecting command characters is a common method of customization.

In the rest of this manual, we usually ignore this subtlety to keep things simple. To give the extension-writer the information he needs, we state the name of the function that really does the work in parentheses after mentioning the command name. For example, we will say that “**C-N** (**Down Real Line**) moves point vertically down,” even though the real truth is that the command function **Down Real Line** is programmed to move point vertically down and **C-N** is initially set up to invoke that function.

While we are on the subject of customization information that you should not be frightened of, it's a good time to tell you about *variables*. Often the description of a command will say, "To change this, set the variable `zwei:*mumble-foo*`." A variable is a name used to remember a value. Most ZMACS variables exist just to permit customization: the variable's value is examined by some command, and changing the value makes the command behave differently. Until you are interested in customizing, you can ignore this information. When you are ready to be interested, read the basic information on variables, and then the information on individual variables will make sense. See section 31.2 [Variables], page 192.

2.4 ZWEI

ZMACS is based the ZWEI editor system (Zwei Was Eine Initially). The name ZWEI refers to a collection of LISP data structures and subroutines for doing editing; ZMACS is the specific program that uses ZWEI to provide a facility for editing files. Two other programs, ZMail and Converse, also use the ZWEI editing routines, for editing incoming mail and communicating interactively with other users. At some time in the future, input editing in LISP Listeners may use another interface to ZWEI.



3. Invoking ZMACS

Since ZMACS is a window-oriented program, the usual way to invoke it is to select the window ZMACS uses (the ZMACS frame). This can be done by clicking on it with the mouse, or using the system menu's Select option, just as you might select any window. Since ZMACS is used so often, there are other convenience methods of selecting the ZMACS frame:

Type **(SYSTEM) E**. This selects an existing ZMACS frame. If one ZMACS frame is already selected, and there are others, this selects one of the others.

Type **(SYSTEM) Control-E**. This creates a new ZMACS frame and selects it.

Click on the system menu's Edit option.

There are also LISP functions for invoking ZMACS:

ed <i>arg</i>	Function
Selects a ZMACS frame that is awaiting a command, creating one if necessary, and then uses <i>arg</i> as a directive to find some text to edit. <i>arg</i> may be	

nil to leave selected whatever buffer happens already to be selected in that ZMACS.

A file name string or pathname, to visit the specified file.

A symbol, to visit the LISP definition of that symbol as if the **Meta-** command were used (see section 26.1 [Sectionization], page 153).

t to create and select a new non-file buffer.

The symbol **zwei:reload** to cause ZMACS to reinitialize its data structure of buffers. Since this causes all existing ZMACS buffers to be lost, it is a very drastic last resort. If you find it necessary to do this, you can probably manage to avoid losing your previous work by calling **zwei:save-all-files** first.

zwei:edit-functions <i>&rest functions</i>	Function
Selects a ZMACS frame that is awaiting a command and then creates a possibilities list (see section 26.7 [Possibilities Lists], page 164) containing the functions <i>function</i> so that the command C-Shift-P can be used to find the definitions of those functions.	

dired <i>directory</i>	Function
Invokes the Dired subsystem of ZMACS. First it selects a ZMACS frame that is awaiting	

a command, creating one if necessary. Then it executes **M-X Dired** on *directory*. See chapter 18 [Dired], page 91.

mail &optional *recipient text call-editor-anyway*. Function
 If two args are specified, **mail** just sends a message with no interaction. If either *recipient* or *text* is **nil**, or if *call-editor-anyway* is non-**nil**, **mail** selects a ZMACS frame that is awaiting a command and then executes **C-X M** in it. If either *text* or *recipient* was specified, it is inserted into the buffer to be used when you eventually send the message. See section 29.2 [Mail], page 179.

call-editor-anyway may be a number; in that case, it is a character position within *text*, saying where point should be positioned inside the mail sending buffer.

swel:load-file-into-ZMACS *file-name* Function
 Creates a ZMACS buffer and visits file *file-name* in it. The new buffer, once created, can be selected in any ZMACS frame, but **zwei:load-file-into-ZMACS** does not affect any particular ZMACS frame. It just makes the buffer available for selection later.

swel:load-directory-into-ZMACS *directory* Function
 Creates a new ZMACS Dired buffer and initializes it to describe directory *directory*. The buffer is then available for selection in any ZMACS frame.

look for a ZMACS frame that is awaiting a command, it is safe to use them, or the debugger commands **Control-E** or **Control-M**, inside a ZMACS frame. The crashed ZMACS frame will not be used since it is not awaiting a command; therefore, some other ZMACS frame will be used, possibly a newly created one. In any case, the state of the ZMACS frame being debugged will not be wiped out.

3.1 Exiting ZMACS

In the LISP machine system, a window-oriented program does not really need a command to exit. When you wish to use some other program, simply select another window. However, there are a few commands in ZMACS that do exit, by selecting the previously selected window.

- C-Z** Exit from ZMACS.
- M-Z** Compile the current buffer and exit from ZMACS.

C-M-Z Evaluate the current buffer and exit from ZMACS.

Exiting ZMACS is much like the **TERMINAL S** command for switching windows, but there is one difference: **TERMINAL S** is executed immediately when you type it, but an exit command such as **C-Z** does not do its work until ZMACS is ready for it. You can type ahead input following an exit command, while ZMACS is doing work such as a compilation, and be sure that the further input will not be read until ZMACS's current activity is finished. With **TERMINAL S**, you would switch windows instantly and ZMACS's activity would continue in the background; then you might not be able to tell when it was finished.

See section 26.3 [Compile Text], page 157, for more information on compilation and evaluation from ZMACS buffers, including the **M-Z** and **C-M-Z** commands.



4. Basic Editing Commands

We now give the basics of how to enter text, make corrections, and save the text in a file. If this material is new to you, you might learn it more easily by running the TEACH-ZMACS learn-by-doing tutorial. (To do this, type **Meta-X Teach ZMACS** **(RETURN)**.)

4.1 Inserting Text

To insert printing characters into the text you are editing, just type them. Except in special modes, ZMACS defines each printing character as a command to insert that character into the text at the cursor (that is, at *point*; see chapter 1 [Screen], page 5). The cursor moves forward. Any characters after the cursor move forward too. If the text in the buffer is 'FOOBBAR', with the cursor before the 'B', then if you type **XX**, you get 'FOOXXBAR', with the cursor still before the 'B'.

To correct text you have just inserted, you can use **(RUBOUT)** (which runs the command function named **Rubout**). **(RUBOUT)** deletes the character *before* the cursor (not the one that the cursor is on top of or under; that is the character *after* the cursor). The cursor and all characters after it move backwards. Therefore, if you type a printing character and then type **(RUBOUT)**, they cancel out.

To end a line and start typing a new one, type **(RETURN)** (running the command function **Insert Crs**). **(RETURN)** operates by inserting a newline (a **(RETURN)** character) in the buffer. If point is in the middle of a line, **(RETURN)** splits the line. Typing **(RUBOUT)** when the cursor is at the beginning of a line rubs out the newline before the line, thus joining the line with the preceding line.

If you add too many characters to one line, without breaking it with a **(RETURN)**, the line will grow to occupy two (or more) lines on the screen, with a '!' at the extreme right margin of all but the last of them. The '!' says that the following screen line is not really a distinct line in the text, but just the *continuation* of a line too long to fit the screen. Sometimes it is nice to have ZMACS insert newlines automatically when a line gets too long; for this, use Auto Fill mode (see section 24.6 [Filling], page 130).

Direct insertion works for printing characters and space, but other characters act as editing commands and do not insert themselves. If you need to insert a nongraphic character such as **(ALTMODE)**, **(END)** or **(ABORT)**, you must *quote* it by typing **Control-Q (Quoted Insert)** first. There are two ways to use it.

Control-Q followed by any non-graphic character (even **ABORT**) inserts that character.

Control-Q followed by three octal digits inserts the character with the specified character code.

A numeric argument to **C-Q** specifies how many copies of the quoted character should be inserted.

In the ASCII character set, control characters are codes 0 through 37 (octal), and are named after the characters with codes 100 through 137 (mostly letters). Thus, code 1 is ASCII control-A. **C-Q** followed by a **Control-** letter inserts the character code for the ASCII control character for that letter. Thus, **C-Q C-A** is equivalent to **C-Q 0 0 1**. This inserts a ↓, which is code 1 in the LISP Machine character set, but is **CTRL-A** when stored on file servers that use ASCII.

4.2 Changing the Location of Point

To do more than insert characters, you have to know how to move point (see section 1.1 [Point], page 5). Here are a few of the commands for doing that.

- C-A** Move to the beginning of the line (**Beginning Of Line**).
- C-E** Move to the end of the line (**End Of Line**).
- C-F** Move forward one character (**Forward**).
- C-B** Move backward one character (**Backward**).
- C-N** Move down one line, vertically (**Down Real Line**). If you start in the middle of one line, you end in the middle of the next. From the last line of text, **C-N** creates a new line and moves onto it.
- C-P** Move up one line, vertically (**Up Real Line**).
- C-L** Clear the screen and reprint everything (**Recenter Window**).
- C-T** Transpose two characters, the ones before and after the cursor (**Exchange Characters**).
- M-<** Move to the top of the buffer (**Goto Beginning**). With numeric argument *n*, move to *n*/10 of the way from the top. See chapter 6 [Arguments], page 27, for more information on numeric arguments.
- M->** Move to the end of the buffer (**Goto End**).
- arg C-M-#** Move to *arg* characters from the top of the buffer (**Goto Character**). See chapter 6 [Arguments], page 27, for information on how to type the argument *arg*. Given a negative argument *-arg*, move to *arg* characters from the top of the buffer, counting each newline as two characters. This is useful for going to a character position identified by a program running on a PDP-10.

C-X C-N Set current column as goal column for **C-N** and **C-P**. Henceforth, those commands move to this fixed column in the line moved to (**Set Goal Column**).

C-U C-X C-N

Cancel the goal column. Henceforth, **C-N** and **C-P** try to stay in the same column, as usual.

4.3 Erasing Text

RUBOUT Delete the character before the cursor (**Rubout**).

C-D Delete the character after the cursor (**Delete Forward**).

C-K Kill to the end of the line (**Kill Line**).

You already know about the **RUBOUT** command which deletes the character before the cursor. Another command, **Control-D**, deletes the character after the cursor, causing the rest of the text on the line to shift left. If **Control-D** is typed at the end of a line, that line and the next line are joined together.

To erase a larger amount of text, use the **Control-K** command, which kills a line at a time. If **Control-K** is done at the beginning or middle of a line, it kills all the text up to the end of the line. If **Control-K** is done at the end of a line, it joins that line and the next line.

See section 11.1 [Killing], page 45, for more flexible ways of killing text.

4.4 Files

The commands above are sufficient for creating and altering text in an ZMACS buffer; the more advanced ZMACS commands just make things easier. But to keep any text permanently you must put it in a *file*. Files are named units of text that are stored on computers called *file servers*. To look at or use the contents of a file in any way, including editing the file with ZMACS, you must specify the file name.

Consider a file named `/usr/rms/foo.bar` residing on a Unix file server named `plugh`. To edit this file, type the command **C-X C-F** and then give the file name `plugh:/usr/rms/foo.bar` as an *argument*, ending it with **RETURN**. ZMACS obeys by *visiting* the file: creating a buffer, copying the contents of the file into the buffer, and then displaying the buffer for you to edit. You can make changes in it, and then *save* the file by typing **C-X C-S**. This makes the changes permanent by

copying the altered contents of the buffer back into the file `/usr/rms/foo.bar` on the server. Until then, the changes are only inside your ZMACS, and the file `foo.bar` is not changed.

To create a file, just visit the file with `C-X C-F` as if it existed. ZMACS will make an empty buffer in which you can insert the text you want to put in the file. When you save your text with `C-X C-S`, the file will be created.

Of course, there is a lot more to learn about using files. See chapter 17 [Files], page 75.

4.5 Help

If you forget what a command does, you can find out with the `(HELP)` character. Type `(HELP)` followed by `c` and the command you want to know about. `(HELP)` can help you in other ways as well. See chapter 9 [Help], page 37.

4.6 Blank Lines

C-0 Insert one or more blank lines after the cursor.

C-X C-0 Delete all but one of many consecutive blank lines.

When you want to insert a new line of text before an existing line, you can do it by typing the new line of text, followed by `(RETURN)`. However, it may be easier to see what you are doing if you first make a blank line and then insert the desired text into it. This is easy to do using the command `C-0` (Customizers: this is the function **Make Room**), which inserts a newline after point but leaves point in front of the newline. After `C-0`, type the text for the new line. `C-0 F 0 0` has the same effect as `F 0 0 (RETURN)`, except for the final location of the cursor.

You can make several blank lines by typing `C-0` several times, or by giving it an argument to tell it how many blank lines to make. See chapter 6 [Arguments], page 27, for how.

If you have many blank lines in a row and want to get rid of them, use the command `C-X C-0` (the command function **Delete Blank Lines**). When point is on a blank line that is adjacent to at least one other blank line, `C-X C-0` deletes all but one of the consecutive blank lines, leaving exactly one. With point on a blank line with no other blank line adjacent to it, the sole blank line is deleted, leaving none. When point is on a nonblank line, `C-X C-0` deletes any blank lines following that nonblank line.

5. Basic Mouse Commands

The mouse can be used to move the ZMACS cursor, to scroll text in the window, and also to kill, move or copy text.

5.1 Cursor Motion and Regions

Moving the cursor is done with \mathbb{L} . Point moves to the nearest possible spot to where the mouse is. (Remember that point can be located only next to a character that exists; it cannot be located past the end of the text on a line.)

If you press \mathbb{L} and move the mouse while holding the button down, you will mark a region. Mark will be at the place where you pushed the button down, and point where you let go of the button. As you move the mouse, the underlining changes to show you what region you will select. If you click without moving the mouse, point and mark are set at the same place. See chapter 10 [Mark], page 39.

You can also mark a region using \mathbb{M} . Depending on the position of the mouse, and on the ZMACS major mode, you can mark a word, line, sentence, or LISP expression.

At the beginning or end of a line, the entire line is marked (not including the following newline).

At a parenthesis, the entire parenthetical grouping (including parentheses) is marked.

At a double-quote (""), the entire quotation or string is marked.

Anywhere else, the word or LISP symbol that point is in is marked. (LISP symbols are marked in LISP mode and related modes).

If you move the mouse with the middle button held down, the underlined textual unit changes as the mouse moves. When you release the mouse button, the text then underlined becomes the new region.

\mathbb{L} moves the mouse cursor instantaneously to where point is. This is useful if you wish to use the mouse for editing operations in the vicinity of point, because you do not have to move the mouse way across the screen to get there. \mathbb{L} has no effect on point or on the text being edited.

5.2 Killing and Yanking

Killing and yanking with the mouse are done with \mathbb{M} . This one command can kill or yank, depending on circumstances. See section 11.1 [Killing], page 45.

- If there is a nonempty region, then its contents are copied into the kill history. This is like the **Meta-W** command.
- If \mathbb{M} is done a second time, with no other commands in between, then the region that was saved the first time is now deleted from the text.
- In any other circumstances, the contents of the top of the kill history are inserted in the buffer at point. This is just like the **C-Y** command.

Note that the position of the mouse does not matter when \mathbb{M} is done. The effect of the command depends only on point, the mark, and the kill history.

5.3 Scrolling

To scroll with the mouse, use the scroll bar at the left edge of the ZMACS window. Push the mouse against the left edge of the window until the mouse cursor changes shape to a thick up-and-down arrow. The standard scroll bar commands are available:

\mathbb{L} moves the line the mouse is pointing at to the top of the window.

\mathbb{B} moves that line to the bottom of the window.

\mathbb{R} moves the line at the top of the window down to where the mouse points.

\mathbb{M} uses the position of the mouse down the side of the window to specify which part of the buffer should be visible. The mouse at the top of the window means show the start of the buffer; the bottom of the window means show the end of the buffer; 1/3 of the way down the window means show the text 1/3 of the way into the buffer.

Another way to scroll with the mouse is to push it against the top or bottom edge of the text window, near the right edge. If the mouse cursor changes to a thick arrow, you have found the right spot. As you push the mouse, the buffer scrolls one line at a time.

5.4 Mouse-Sensitive Typeout

Typeout often contains mouse-sensitive text that does something if you click the mouse on it.

This is implemented using the flavor `tv:basic-mouse-sensitive-items`; refer to the *Window System Manual* for information on it. A mouse-sensitive text item displays a rectangular outline when the mouse is inside it; this is how you can tell that you have found mouse-sensitive typeout.

Typically each mouse-sensitive item defines several operations; \uparrow performs one of them, the one we expect you will use most often, while \uparrow pops up a menu of all the defined operations. When the mouse is pointing at the item, the mouse documentation line at the bottom of the screen lists all the available operations.

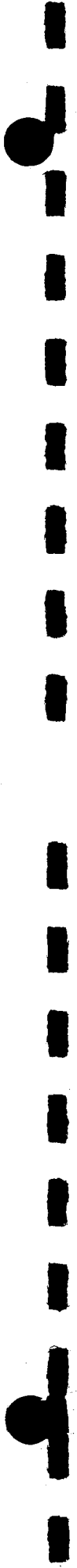
For example, each line in the list of buffers printed by `C-X C-B` is mouse-sensitive, and clicking on it operates on the buffer described on that line. The most common operation is to select the buffer; \uparrow does this. \uparrow brings up a menu containing both selection and other operations such as killing and saving.

5.5 The Editor Menu

The command \uparrow in the text window invokes a pop-up menu of editor commands. Most of these commands can be invoked with characters or extended commands also, and act no differently when invoked with the menu.

5.6 Arguments

Commands that read the name of a LISP function using the minibuffer provide a special meaning for \uparrow when the minibuffer is empty. All names of LISP functions in the text become mouse-sensitive and are outlined when pointed at. \uparrow causes the LISP function name pointed at to be used as the argument to the command. The mouse cursor changes to a thin arrow pointing straight upward when this capability is available.



6. Numeric Arguments

Any ZMACS command can be given a *numeric argument*. Some commands interpret the argument as a repetition count. For example, giving an argument of ten to the **C-F** command (move forward one character) moves forward ten characters. With these commands, no argument is equivalent to an argument of one. Negative arguments are allowed. Often they tell a command to move or act backwards.

Some commands care only about whether there is an argument, and not about its value. For example, the command **M-Q (Fill Paragraph)** with no argument fills text; with an argument, it justifies the text as well. (See section 24.6 [Filling], page 130, for more information on **M-Q**.)

Some commands use the value of the argument as a repeat count, but do something peculiar when there is no argument. For example, the command **C-K (Kill Line)** with argument *n* kills *n* lines, including their terminating newlines. But **C-K** with no argument is special: it kills the text up to the next newline, or, if point is right at the end of the line, it kills the newline itself. Thus, two **C-K** commands with no arguments can kill a nonblank line, just like **C-K** with an argument of one. (See section 11.1 [Killing], page 45, for more information on **C-K**.)

The simplest way to specify an argument is to type digits and/or a minus sign while holding down the **CONTROL** key, the **META** key, the **HYPER** key, or the **SUPER** key, or any combination of them. It is easiest to use the same combination of shift keys for the digits that you are going to use for the command that follows; thus, **Control-5 Control-N** or **Control-Meta-Minus Control-Meta-8 Control-Meta-U**.

Another way of specifying an argument is to use the **C-U (Universal Argument)** command followed by the digits of the argument. With **C-U**, you can type the argument digits without holding down shift keys. To type a negative argument, start with a minus sign. Just a minus sign normally means -1.

C-U followed by a character that is neither a digit nor a minus sign has the special meaning of "multiply by four". It multiplies the argument for the next command by four. Two such **C-U**'s multiply it by sixteen. Thus, **C-U C-U C-F** moves forward sixteen characters. This is a good way to move forward "fast", since it moves about 1/5 of a line in the usual size window and font. Other useful combinations are **C-U C-N**, **C-U C-U C-N** (move down a good fraction of a screen), **C-U C-U C-O** (make "a lot" of blank lines), and **C-U C-K** (kill four lines). With commands like **M-Q** that care whether there is an argument but not what the value is, **C-U** is a good way of saying, "Let there be an argument."

A few commands treat a plain **C-U** differently from an ordinary argument. A few others may treat an argument of just a minus sign differently from an argument of **-1**. These unusual cases will be described when they come up; they are always for reasons of convenience of use of the individual command.

7. The Minibuffer

The *minibuffer* is the facility used by ZMACS commands to read arguments more complicated than a single number. Minibuffer arguments can be file names, buffer names, LISP function names, ZMACS command function names, LISP expressions, and many other things, depending on the command reading the argument.

The minibuffer appears in the echo area, with a rectangular outline. A blinking cursor appears within it. The cursor in the editing window above stops blinking, because that window is no longer selected. Above the minibuffer, the mode line displays a prompt describing what kind of argument you should enter with the minibuffer, and how; and what defaults there are, if any. This is instead of the usual contents of the mode line.

Usually, you will enter a minibuffer argument by typing the text and ending with **(RETURN)** or **(END)**. In some case, where multi-line arguments are expected, only **(END)** can be used to exit; the mode line always says so when this is so. When completion is available, these two characters both exit but do other things differently; see below. The mode line says when completion is available.

The command **(ABORT)** is defined, in the minibuffer, to get out of the minibuffer and cancel the command for which you were supplying an argument. **Control-G** has the same effect, if the minibuffer is empty; if it is not empty, **Control-G** deletes its contents, so that one or two **Control-G** commands has the same effect as **(ABORT)**.

The minibuffer is a ZMACS buffer (albeit a peculiar one), and the usual ZMACS commands are available for editing the text of an argument you are editing. You can even switch to another window using **C-X O**, and change the text there, before returning to the minibuffer to submit the argument. You can kill text in another window, return to the minibuffer window, and then yank the text to use it in the argument. But you cannot use any commands that need the minibuffer while the minibuffer is in use; recursive minibuffers are not allowed. Also, you cannot switch buffers in the minibuffer window, even with a command like **C-M-L** that does not use the minibuffer itself. The minibuffer and its window are permanently attached. You can switch buffers using **C-M-L** or mouse clicks in other windows while the minibuffer is active, however.

Often there is a default argument which is used if you type **(RETURN)** without inserting any text. The prompt line says what the default is. The command **C-Shift-Y** (**Yank Default String**) inserts the text for the default into the minibuffer so you can use it with modifications. **C-Shift-Y** is defined this way only in the minibuffer.

Histories are kept for many kinds of minibuffer arguments. For example, all buffer name ar-

arguments go on one history, all file name arguments go on another, and all LISP function name arguments go on yet another history. The contents of the appropriate history can be yanked into the minibuffer with **C-M-Y** (**Yank Previous Input**). Numeric arguments can be used to refer to earlier entries in the history, as with **C-Y**, and **M-Y** can be used to move around in the history. **C-M-O** **C-M-Y** prints a list of the recent elements in the history. See section 11.2 [Yanking], page 47. **C-M-Y** is defined this way only in the minibuffer.

You can also yank the last string that **C-S** searched for. To do this, type **C-Shift-S** (**Yank Search String**). **C-Shift-S** is defined this way only in the minibuffer.

$\left(\frac{1}{2}\right)$ when the mouse is pointing at a non-completing minibuffer window exits the minibuffer, just like **END**.

7.1 Completion

Often, the minibuffer provides a *completion* facility. This means that you type enough of the argument to determine the rest, based on ZMACS's knowledge of which arguments make sense, and ZMACS visibly fills in the rest, or as much as can be determined from the part you have typed.

When completion is available, certain commands—**ALTMODE**, **END**, **RETURN**, and **SPACE**—are redefined to complete an abbreviation present in the minibuffer into a longer string that it stands for, by matching it against a set of *completion alternatives* provided by the command reading the argument. The word '**completion**' appears in parentheses at the right hand edge of the mode line when completion is available.

For example, when the minibuffer is being used by **Meta-X** to read the name of a ZMACS extended command, it is given a list of all available ZMACS extended command names to complete against. The completion characters match the text in the minibuffer against all the command names, find any additional characters of the name that are implied by the ones already present in the minibuffer, and add those characters to the ones you have given.

Here is a list of all the completion commands, defined in the minibuffer when completion is available.

- ALTMODE** Complete the text in the minibuffer as much as possible.
- SPACE** Complete the text in the minibuffer but don't add or fill out more than one word.
- RETURN** Submit the text in the minibuffer as the argument, possibly completing first as de-

scribed below.

C-RETURN

Submit the text in the minibuffer as the argument, with no completion.

END

Complete the text in the minibuffer as much as possible, and if the result is an exact match, submit it as the argument.

C-?

Print a list of all possible completions of the text in the minibuffer. If there is only one, some documentation of it may also be printed. The possible completions are mouse-sensitive; clicking **[L]** on one of them chooses it as the argument and exits the minibuffer immediately.

C-/

Print a list of all completion alternatives containing any one or more of the words in the minibuffer. They are mouse-sensitive like the completions **C-?** prints.

Completion is very hard to explain but easy to understand once you have seen it in operation. If you type **Meta-X a SPACE f SPACE**, the second **SPACE** looks for an alternative (in this case, an extended command name) whose first word starts with **a** and whose second word starts with **f**. There is only one such command name: **Auto Fill Mode**. So the **'a'** is changed to **'Auto'** and the **'f'** is changed to **'Fill'**. So at this point the minibuffer contains **'Auto Fill'**. If you type another **Space**, it inserts the word **'Mode'**, since that is the only possible third word that makes a valid extended command name.

The **ALTMODE** command performs completion much like **SPACE**, but it does not stop at the end of a word. It adds on as many words or parts of words as it can determine from what you have typed. If you type **Meta-X a SPACE f ALTMODE, ALTMODE** not only extends the **'a'** to **'Auto'** and the **'f'** to **'Fill'**, it adds the word **'Mode'** immediately.

The **END** command which exits the minibuffer also does completion just like **ALTMODE**. If your input completes to a unique alternative, **END** exits the minibuffer. If your input is an abbreviation for two different alternatives, or is not an abbreviation for any, then **END** just beeps, and the minibuffer remains displayed so you can correct your input and try **END** again.

There are three different ways that the **RETURN** command can work in completing minibuffers, depending on how the argument will be used.

Strict completion is used when it is meaningless to give any argument except one of the known alternatives. For example, when **C-X K** reads the name of a buffer to kill, it is meaningless to give anything but the name of an existing buffer. In strict completion, **RETURN** refuses to exit if the text in the minibuffer does not complete to an exact match.

Permissive completion is used when any string whatever is meaningful, and the list

of completion alternatives is just a guide. For example, when **C-X C-F** reads the name of a file to visit, any file name is allowed, in case you want to create a file. In permissive completion, **(RETURN)** takes the text in the minibuffer exactly as given, without completing it.

Reluctant completion is used only by **C-X B** to read the name of a buffer to switch to. In this case, **(RETURN)** attempts to match the text in the minibuffer as an abbreviation for a buffer name. If it matches, the buffer name matched is used as the argument. If it does not match, the first **(RETURN)** just beeps, but if you type **(RETURN)** again immediately after, it uses the text as present in the buffer as the argument. The character **C-(RETURN)** can be used to force exiting without even trying completion.

(SPACE) and **C-/** are not available for completion of file names. This is because file names are completed by file servers, not by ZMACS.

Two mouse commands are different when the mouse is pointing at a completing minibuffer: **(L)** and **(R)**. **(L)** has the same effect as typing **(RETURN)**. **(R)** pops up a menu containing a list of possible completions of the text already in the minibuffer, so that you can then use the mouse to choose one. Choosing from the menu exits the minibuffer immediately, returning the chosen completion.

7.1.1 Matching for Completion

When text in the minibuffer is to be matched against a completion alternative, both are first broken up into words. Each successive word of the minibuffer text must match the beginning of the corresponding word of the completion alternative, in order to make that alternative a possible completion.

Once all the possible completions of the minibuffer text are found, they are compared word by word. In each word, all the characters common to the beginnings of all the possible completions become the completion result.

Most completion commands insert the entire completion result into the minibuffer in place of the existing text. This has the effect of inserting any characters that are uniquely determined by the characters already there.

(SPACE) works differently. It inserts only as many words of the completion result as there were word separators in the minibuffer already (including the **(SPACE)** just inserted). The result is that each time you type a **(SPACE)**, only one word is added to the minibuffer. The **(SPACE)** characters you type match the spaces in the completion alternative you are abbreviating.

7.2 Repeating Minibuffer Commands

Every command that uses the minibuffer at least once is recorded on a special history list, together with the values of the minibuffer arguments, so that you can repeat the command easily. In particular, every **Meta-X** command is recorded, since **M-X** uses the minibuffer to read the command name.

C-X **(ALTMODE)**

Re-execute a recent minibuffer command.

M-Shift-Y

Re-execute a different command instead.

C-X **(ALTMODE)** (**Repeat Mini Buffer Command**) is used to re-execute a recent minibuffer-using command. With no argument, it repeats the last such command. A numeric argument larger than one specifies an earlier command to repeat.

The previous command is repeated as if you had typed it again; but each time it needs to read an argument using the minibuffer, the minibuffer starts out containing the same text that was supplied the last time the command was run. If you want to use the same argument as before, exit immediately with **(RETURN)** or **(END)** as appropriate; otherwise, edit the argument first.

If **C-X** **(ALTMODE)** begins repeating a command other than the one you intended, you can switch to an earlier command by typing **M-Shift-Y** (**Pop Mini Buffer History**). It cancels the command being repeated and starts repeating the previous one in the history. With a numeric argument, it can move any number of commands; a negative argument moves to more recent commands.

M-Shift-Y has another use. If a command reads more than one argument using the minibuffer, **M-Shift-Y** backs up to the previous minibuffer argument, giving you a chance to alter it and proceed again.

C-U O C-X **(ALTMODE)** prints a list of the previous minibuffer commands now remembered. Each one is preceded by a number giving its position in the list—the argument you would have to give to **C-X** **(ALTMODE)** to repeat that command. You can also repeat any of those commands by clicking **(L)** on the line that describes it.



8. Extended (Meta-X) Commands

Not all ZMACS commands are of the one or two character variety you have seen so far. Other commands are invoked by long names composed of English words, such as **Auto Fill Mode** or **Fill Long Comment**. This is because the long names are easier to remember, for commands that are not used often. The commands with long names are known as *extended commands* because they extend the set of two-character commands. This chapter tells more about how to type extended commands in general. Individual extended commands are documented throughout the manual.

Extended commands are also called *M-X commands*, because they consist of **Meta-X (Extended Command)** followed by the name of the extended command. **M-X** reads the command name using the minibuffer. Terminate the command name with **(RETURN)**. For example, the extended command **Fill Long Comment** can be invoked by typing

M-X Fill Long Comment (RETURN)

ZMACS uses the minibuffer for reading input for many different purposes; on this occasion the mode line, which is just above the box, changes to display the string '**Extended command:**' as a *prompt* to tell you that on this occasion your input is the name of an extended command. See chapter 7 [Minibuffer], page 29, for full information the features of the minibuffer.

Abbreviated command names are allowed, because *completion* is provided from the set of all possible extended command names. See section 7.1 [Completion], page 30.

If while in the minibuffer you change your mind about issuing an extended command, type **(ABORT)**. This makes the minibuffer disappear and cancels the **M-X** command. It leaves you at top level, with ZMACS ready to read a new command.

Some extended commands can use numeric prefix arguments. Simply give the **Meta-X** command an argument and **Meta-X** will pass it along to the function that it calls. The argument appears in the prompt while the command name is being read.

Normally, when describing an extended command, we omit the **(RETURN)**. After all, it is just a command to exit the minibuffer, and you could alternatively accomplish that using **(END)** or **C-(RETURN)**. We mention the **(RETURN)** only when there is a need to emphasize its presence, such as when describing a sequence of input that contains an extended command and arguments that follow it.

Recall that every ZMACS command invokes a command function that has a multi-word command name. In an extended command, you type the command name itself to say which command function you want to run. Thus, we speak of the command **M-X Save All Files** which runs the command function **Save All Files**. However, not all command functions are available through **M-X**. The **comtabs** control which are available. See section 31.4 [Comtabs], page 199.

9. Help

ZMACS provides extensive self-documentation features which revolve around a single character, **(HELP)**. At any time while using ZMACS, you can type the **(HELP)** character to ask for help.

If you type **(HELP)** in the middle of a multi-character command, it often tells you about what sort of thing you should type next. For example, if you type **M-X** and then **(HELP)**, it tells you how to use the minibuffer and what command names are available. If you are in the minibuffer entering an argument to a command, **(HELP)** prints information on how to use the minibuffer and on the command that is reading the argument.

But normally, when it's time for you to start typing a new command, **(HELP)** offers you several options for asking about what commands there are and what they do. It prompts with a string

Help. Options are C,D,L,A,U,V,W,<Space>,<Help>

and you should type one of those characters. Typing **(HELP)** at this time prints a description of what the options mean. The ones you are likely to need are described here.

The most basic **(HELP)** options are **(HELP) C** and **(HELP) D**. **(HELP) C** describes the meaning of a character or sequence of characters; type it followed by a command sequence, and it prints the documentation of the command function that command sequence runs. **(HELP) D** is similar but reads the name of a command function. Thus, both **(HELP) C M-F** and **(HELP) D Forward Word (RETURN)** print something like "Moves forward by words."

A more complicated sort of question to ask is, "What are the commands for working with files?" For this, type **(HELP) A file (RETURN)**, which prints a list of all command names that contain 'file', such as **Save All Files**, **Find File**, and so on. With each command name appears a brief description of how to use the command, and what character sequence you can currently invoke it with. For example, it would say that you can invoke **Save File** by typing **C-X C-S. A** stands for 'Apropos'; **(HELP) A** runs the ZMACS command function **Apropos**, which does something analogous to the LISP function **apropos**.

Because **Apropos** looks only for functions whose names contain the string that you specify, you must use ingenuity in choosing substrings. If you are looking for commands for killing backwards and **(HELP) A Kill Backwards** doesn't reveal any, don't give up. Try just 'kill', or just 'backwards', or just 'back'. Be persistent. Pretend you are playing Adventure.

Here is a set of Apropos strings that covers many classes of ZMACS commands, since there are strong conventions for naming the standard ZMACS commands. By giving you a feel for the naming conventions, this set should also serve to aid you in developing a technique for picking Apropos strings.

character, line, word, sentence, paragraph, region, page, buffer, screen, window, file, dir, beginning, end, case, mode, forward, backward, next, previous, up, down, search, kill, delete, mark, fill, indent, change.

If something surprising happens, and you are not sure what commands you typed, use **(HELP)** L. **(HELP)** L prints the last 60 command characters you typed in. If you see commands that you don't know, you can use **(HELP)** C to find out what they do.

To find out about the other **(HELP)** options, type **(HELP)** **(HELP)**. That is, when the first **(HELP)** asks for an option, type **(HELP)** to ask for assistance.

10. The Mark and the Region

There are many ZMACS commands that operate on an arbitrary contiguous part of the current buffer. To specify the text for such a command to operate on, you set *the mark* at one end of it, and move point to the other end. The text between point and the mark is called *the region*, and is underlined on the screen. You can move point or the mark to adjust the boundaries of the region. It doesn't matter which one is set first chronologically, or which one comes earlier in the text.

After creating a region, normally you will issue a command that operates on the region. Most such commands *clear out* or *deactivate* the mark. Once this happens, there is no mark, and no region, and no underlining, until you set the mark again. However, the former location of mark is still remembered, and you can reactivate it with **C-X C-X** if you want to use the same region again. Most commands that change the text also deactivate the region even if they do not use it; this includes typing text to insert it. **C-G** and **(ABORT)** also deactivate the region.

Some commands that insert large amounts text, such as **M-X Insert Buffer**, position the mark at one end of the inserted text but do not activate it. They do not create a region, but they make it easy to select the text just inserted as a region if you want to do so.

Here are some commands for setting the mark:

- C-(SPACE)** Set the mark where point is.
- C-@** The same.
- C-X C-X** Interchange mark and point.
- M-@** Set mark after end of next word. This command and the following three do not move point.
- C-M-@** Set mark after end of next LISP expression.
- C-<** Set mark at beginning of buffer.
- C->** Set mark at end of buffer.
- M-H** Put region around current paragraph.
- C-M-H** Put region around current LISP defun.
- C-X H** Put region around entire buffer.
- C-X C-P** Put region around current page.

For example, if you wish to convert part of the buffer to all upper-case, you can use the **C-X C-U** command, which operates on the text in the region. You can first go to the beginning of the text to be capitalized, type **C-(SPACE)** to put the mark there, move to the end, and then type **C-X**

C-U. Or, you can set the mark at the end of the text, move to the beginning, and then type **C-X C-U**. **C-X C-U**'s command function is **Uppercase Region**, whose name signifies that the region, or everything between point and the mark, is to be capitalized.

The most common way to set the mark is with the **C-SPACE** command (**Set Pop Mark**). This sets the mark where point is. Then you can move point away, leaving the mark behind.

The command **C-X C-X** (**Swap Point and Mark**) puts the mark where point was and point where the mark was. The extent of the region is unchanged, but the cursor and point are now at the previous location of the mark. **C-X C-X** is useful when you are satisfied with the location of point but want to move the mark; do **C-X C-X** to put point there and then you can move it. A second use of **C-X C-X**, if necessary, puts the mark at the new location with point back at its original location.

When the mark is deactivated, you can use **C-X C-X** to reactivate it. It moves point to the inactive location of the mark, and sets an active mark at the former location of point. A second use of **C-X C-X** puts point back at its original location, with the formerly inactive mark now active.

Switching buffers also clears out the mark. Each buffer remembers individually its former mark location, so that **C-X C-X** always resurrects the current buffer's last region.

10.1 Operating on the Region

Once you have created an active region, you can do many things to the text in it:

Kill it with **C-W** (see section 11.1 [Killing], page 45).

Undo changes in it with **C-Shift-U** (see chapter 12 [Undo], page 53).

Save it in a register with **C-X X** (see section 11.3.2 [Registers], page 51).

Save it in a buffer or a file (see section 11.3.1 [Accumulating Text], page 50).

Convert case with **C-X C-L** or **C-X C-U** (see section 23.1 [Case], page 119).

Change fonts in it with **C-Shift-J** or **M-Shift-J** (see section 23.2 [Fonts], page 120).

Compile it as LISP with **C-Shift-C** (see section 26.3 [Compile Text], page 157).

Kill comments in it with **C-M-;** (see section 25.6 [Comments], page 143).

Fill it as text with **M-G** (see section 24.6 [Filling], page 130).

Print hardcopy with **M-X Hardcopy Region** (see section 29.3 [Hardcopy], page 180).

Indent it with **C-X TAB** or **C-M-** (see chapter 22 [Indentation], page 115).

Sort it in various ways (see section 29.4 [Sorting], page 181).

10.2 Commands to Mark Textual Objects

There are commands for placing the mark on the other side of a certain object such as a word or a list, without having to move there first. **M-@ (Mark Word)** puts the mark at the end of the next word, while **C-M-@ (Mark Sexp)** puts it at the end of the next LISP expression. **C-> (Mark End)** puts the mark at the end of the buffer, while **C-< (Mark Beginning)** puts it at the beginning. These characters allow you to save a little typing or redisplay, sometimes.

Other commands set both point and mark, to delimit an object in the buffer. **M-H (Mark Paragraph)** moves point to the beginning of the paragraph that surrounds or follows point, and puts the mark at the end of that paragraph. **M-H** does all that's necessary if you wish to indent, case-convert, or kill a whole paragraph. **C-M-H (Mark Defun)** similarly puts point before and the mark after the current or following defun. **C-X C-P (Mark Page)** puts point before the current page (or the next or previous, according to the argument), and mark at the end. The mark goes after the terminating page delimiter (to include it), while point goes after the preceding page delimiter (to exclude it). Finally, **C-X H (Mark Whole)** sets up the entire buffer as the region, by putting point at the beginning and the mark at the end.

Two variables control how the extent of the region is displayed when it is active. **zwei:*region-marking-mode*** specifies whether to underline the text in the region or highlight it with reverse video. The value of the variable is either **:underline** or **:reverse-video**. **zwei:*region-right-marking-mode*** controls what to do with the space after the end of a line when the region extends beyond. Anything but **nil** means to underline or highlight that space; **nil** means to leave it blank. The defaults are **:underline** and **nil**.

10.3 The Point Pdl

Several commands record the location of point in case you wish to move back to that place later. For example, **C-(SPACE)** does this, in addition to setting the mark. Whenever the location of point is saved in this way, the string **'Point Pushed'** is displayed in the echo area.

The recorded locations go on the *point pdl*, which is just a list of saved point locations, kept most recent first. To return to the last saved location, use **C-U C-(SPACE)**. This moves point to the last saved location and puts that saved location at the end of the list. Repeated use of **C-U C-(SPACE)** therefore touches on each of the saved locations in turn, and eventually repeats.

There is only one point pdl, used in common by all buffers. So **C-U C-SPACE** can take you back to a previously selected buffer.

Many commands that can move long distances, such as **M-<** (**Goto Beginning**) and **C-M-A** (**Beginning Of Defun**), save the location of point before moving. This is to make it easier for you to move back later. Searches sometimes record the old point; it depends on how far they move. You can tell that a search has done this because it prints **'Point Pushed'** when exited. The variable **zwei:auto-push-point-option*** specifies how many lines a search must move before pushing the previous value of point.

The variable **zwei:auto-push-point-notification*** specifies the string to use to say when point has been pushed. **'Point Pushed'** is its default value.

The variable **zwei:point-pdl-max*** is the maximum number of entries to keep in the point pdl. If that many entries exist and another one is pushed, the last one in the list is discarded. At any time, repeating the command **C-U C-SPACE** circulates through the limited number of entries that are still retained.

10.4 Named Marks

In addition to the point pdl, which attempts to record automatically recent point locations you might wish to return to, ZMACS provides commands with which you can record locations in *registers* and return to them after any amount of time. The name of a register is a single graphic character (letter, digit or punctuation). A location saved in a register is also called a *named mark*.

C-X S r Save location of point in register *r*.

C-X J r Jump to the location saved in register *r*.

To save the current location of point in a register, choose a name *r* and type **C-X S r**. (**C-X S** runs the command function **Save Position In Register**.) The register *r* retains the location thus saved until another **C-X S** command is used to save a different location into that register.

The command **C-X J r** (**Jump to Register Position**) moves point to the location recorded in register *r*. The register is not affected; it continues to record the same location. You can jump to the same position using the same register any number of times.

Registers can also be used to hold text; each register can hold both a saved location and a saved

text string, independently. See section 11.3.2 [Registers], page 51.

M-X List Registers prints a list of all registers that you have used, giving for each one the text it contains (or '[EMPTY]' if none) and the location recorded in it, if any. The text is summarized by showing only the beginning or the end, with text in the middle, and any leading or trailing whitespace, replaced by dots. The location is shown by displaying the text surrounding it, with '-|-' in the middle indicating where the location is.



11. Killing and Moving Text

Killing means erasing text and copying it into the *kill history*, from which it can be retrieved by *yanking* it.

The commonest way of moving or copying text with ZMACS is to kill it and later yank it in one or more places. This is very safe because all the text ever killed is remembered, and it is versatile, because the many commands for killing syntactic units can also be used for moving those units. There are also other ways of copying text for special purposes.

There is only one kill history for the entire LISP Machine system; all buffers use the same one, so you can kill text in one buffer and yank it in another buffer. All editors use it, including ZMACS, ZMail, Converse, and the input editor used in LISP Listener windows. So you can move text between windows by killing in one window and yanking in another.

11.1 Deletion and Killing

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 newlines. Commands that can destroy significant amounts of nontrivial data generally kill. The commands' names and individual descriptions use the words 'kill' and 'delete' to say which they do. If you do a kill or delete command by mistake, you can use the **C-Shift-U** (**Quick Undo**) command to undo it (see chapter 12 [Undo], page 53).

11.1.1 Deletion

- | | |
|----------------|---|
| C-D | Delete next character. |
| RUBOUT | Delete previous character. |
| M-\ | Delete spaces and tabs around point. |
| C-\ | Delete spaces and tabs around point, leaving one space. |
| C-X C-0 | Delete blank lines around the current line. |

M-^ Join two lines by deleting the intervening newline, and any indentation following it.

The most basic delete commands are **C-D (Delete Forward)** and **(RUBOUT) (Rubout)**. **C-D** deletes the character after point, the one the cursor is “on top of”. Point doesn’t move. **(RUBOUT)** deletes the character before the cursor, and moves point back. Newlines can be deleted like any other characters in the buffer; deleting a newline joins two lines. Actually, **C-D** and **(RUBOUT)** aren’t always delete commands; if given an argument, they kill instead, since they can erase more than one character this way.

The other delete commands are those that delete only formatting characters: spaces, tabs and newlines. **M-\ (Delete Horizontal Space)** deletes all the spaces and tab characters before and after point. **C-\ (Just One Space)** does likewise but leaves a single space after point, regardless of the number of spaces that existed previously (even zero).

C-X C-0 (Delete Blank Lines) deletes all blank lines after the current line, and if the current line is blank deletes all blank lines preceding the current line as well (leaving one blank line, the current line). **M-^ (Delete Indentation)** joins the current line and the previous line, or the current line and the next line if given an argument, by deleting a newline and all surrounding spaces, possibly leaving a single space. See chapter 22 [Indentation], page 115.

Several of these commands deal with blanks. The variable **zwei:*blanks*** tells ZMACS which characters to consider blank; its value is a list of characters. By default, the characters in the list are space and tab. Some other commands deal with “whitespace”, which includes newlines. The variable **zwei:*whitespace-chars*** tells ZMACS which characters are whitespace; initially, they are space, tab, newline and backspace.

11.1.2 Killing by Lines

C-K Kill rest of line or one or more lines.

(CLEAR-INPUT)

Kill text on current line up to point.

The simplest kill command is **C-K (Kill Line)**. If given at the beginning of a line, it kills all the text on the line, leaving it blank. If given on a blank line, the blank line disappears. As a consequence, if you go to the front of a non-blank line and type two **C-K**'s, the line disappears completely.

More generally, **C-K** kills from point up to the end of the line, unless it is at the end of a line.

In that case it kills the newline following the line, thus merging the next line into the current one. Invisible spaces and tabs at the end of the line are ignored when deciding which case applies, so if point appears to be at the end of the line, you can be sure the newline will be killed.

If **C-K** is given a positive argument, it kills that many lines and the newlines that follow them (however, text on the current line before point is spared). With a negative argument, it kills back to a number of line beginnings. An argument of -2 means kill back to the second line beginning. If point is at the beginning of a line, that line beginning doesn't count, so **C-U - 2 C-K** with point at the front of a line kills the two previous lines.

C-K with an argument of zero kills all the text before point on the current line. **(CLEAR-INPUT)** (**Clear**) does the same thing.

11.1.3 Other Kill Commands

C-W Kill region (from point to the mark).

M-D Kill word.

M-(RUBOUT)
Kill word backwards.

C-X (RUBOUT)
Kill back to beginning of sentence.

M-K Kill to end of sentence.

C-M-K Kill s-expression.

C-M-(RUBOUT)
Kill s-expression backwards.

A very general kill command is **C-W (Kill Region)**, which kills everything between point and the mark. With this command, you can kill any contiguous sequence of characters, if you first set the mark at one end of them and go to the other end.

Other syntactic units can be killed: words, with **M-(RUBOUT)** and **M-D** (see section 24.2 [Words], page 126); s-expressions, with **C-M-(RUBOUT)** and **C-M-K** (see section 25.2 [Lists], page 136); and sentences, with **C-X (RUBOUT)** and **M-K** (see section 24.3 [Sentences], page 127).

The mouse can also be used to kill. See section 5.2 [Mouse Killing], page 23.

11.2 Yanking

Yanking is getting back text that was killed. The usual way to move or copy text is to kill it and then yank it one or more times.

- C-Y** Yank last killed text.
- M-Y** Replace re-inserted killed text with the previously killed text.
- M-W** Save region as last killed text without actually killing it.
- C-M-W** Append next kill to last batch of killed text.

All killed text is recorded in the *kill history*, a list of blocks of text that have been killed. There is only one kill history for the entire LISP Machine system; all buffers use the same one, so you can kill text in one buffer and yank it in another buffer. All editors use it, including ZMACS, ZMail, Converse, and the input editor used in LISP Listener windows. So you can move text between windows by killing in one window and yanking in another.

The command **C-Y** (**Yank**) reinserts the text of the most recent kill. It leaves the cursor at the end of the text. It sets the mark at the beginning of the text but does not activate it; as a result, there is no region when the **C-Y** command is finished, but the region can be set around the yanked text with **C-X C-X**. See chapter 10 [Mark], page 39.

C-U C-Y leaves the cursor in front of the text, and the inactive mark after it. This is only if the argument is specified with just a **C-U**, precisely. Any other sort of argument, including **C-U** and digits, has an effect described below (under "Yanking Earlier Kills").

If you wish to copy a block of text, you might want to use **M-W** (**Save Region**), which copies the region into the kill history without removing it from the buffer. This is approximately equivalent to **C-W** followed by **C-Y**, except that **M-W** does not mark the buffer as "modified" and does not temporarily change the screen.

The mouse can also be used to yank. See section 5.2 [Mouse Killing], page 23.

11.2.1 Appending Kills

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 entry, so that a single **C-Y** command gets it all back as it was before it was killed. This means that you don't 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. (Thus we join television in leading people to kill thoughtlessly.)

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 entry without rearrangement. Numeric arguments do not break the sequence of appending kills. For example, suppose the buffer contains

```
This is the first
line of sample text
and here is the third.
```

with point at the beginning of the second line. If you type C-K M-2 M-**(RUBOUT)** C-K, the first C-K kills the text 'line of sample text', M-2 M-**(RUBOUT)** kills 'the first' with the newline that followed it, and the second C-K kills the newline after the second line. The result is that the buffer contains 'This is and here is the third.' and a single kill entry contains 'the first**(RETURN)**line of sample text**(RETURN)**'—all the killed text, in its original order.

If a kill command is separated from the last kill command by other commands (not just numeric arguments), it starts a new entry on the kill history. But you can force it to append by first typing the command C-M-W (**Append Next Kill**) in front of it. The C-M-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 entry. With C-M-W, you can kill several separated pieces of text and accumulate them to be yanked back in one place.

11.2.2 Yanking Earlier Kills

To recover killed text that is no longer the most recent kill, you need the **Meta-Y (Yank Pop)** command. The M-Y command should be used only after a C-Y command or another M-Y. It takes the text previously yanked and replaces it with the text from an earlier kill. So, to recover the text of the next-to-the-last kill, you first use C-Y to recover the last kill, and then use M-Y to replace it with the previous kill.

You can think in terms of a "last yank" pointer which points at an item in the kill history. Each time you kill, the "last yank" pointer moves to the newly made item at the front of the history. C-Y yanks the item that the "last yank" pointer points to. M-Y moves the "last yank" pointer to a different item, and the text in the buffer changes to match. Enough M-Y commands can move the pointer to any item in the history, so you can get any item into the buffer. Eventually the pointer

reaches the end of the history; the next **M-Y** moves it to the first item again. (Wrapping from the end to the beginning like this is allowed only when the variable `zwei:*history-yank-wraparound*` is non-`nil`, but that is the default.)

M-Y can take a numeric argument, which tells it how many items to advance the “last yank” pointer by. A negative argument moves the pointer toward the front of the history.

Once the text you are looking for is brought into the buffer, you can stop doing **M-Y**'s and it will stay there. It's just a copy of the history item, so editing it in the buffer does not change what's in the history. As long as no new killing is done, the “last yank” pointer remains at the same place in the kill history, so repeating **C-Y** will yank another copy of the same old kill.

C-Shift-U (Quick Undo) undoes a sequence of **C-Y** and **M-Y**'s all at once: it deletes the yanked text. See chapter 12 [Undo], page 53.

If you know how many **M-Y**'s it would take to find the text you want, you can yank that text in one step using **C-Y** with a numeric argument. **C-Y** with an argument greater than one restores the text the specified number of entries back in the kill history. Thus, **C-U 2 C-Y** gets the next to the last block of killed text. It is equivalent to **C-Y M-Y**. **C-Y** with a numeric argument sets the “last yank” pointer to the entry that it yanks. The argument normally counts relative to the most recent kill, ignoring the previous “last yank” pointer. However, if the variable `zwei:*history-rotate-if-numeric-arg*` is non-`nil`, the argument counts from the “last yank” pointer.

To view the kill history, do **C-Y** with argument zero. The variable `zwei:*history-menu-length*` specifies how many history elements this should show; the default is 20.

11.3 Other Ways of Copying Text

Usually we copy or move text by killing it and yanking it, but there are other ways that are useful for copying one block of text in many places, or for copying many scattered blocks of text into one place.

11.3.1 Accumulating Text

You can accumulate blocks of text from scattered locations either into a buffer or into a file if you like.

C-X A Append region to contents of specified buffer.

M-X Insert Buffer

Insert contents of specified buffer into current buffer at point.

M-X Append To File

Append region to contents of specified file, at the end.

M-X Prepend To File

Append region to contents of specified file, at the beginning.

To accumulate text into a buffer, use the command **C-X A *buffername* (Append To Buffer)**, which inserts a copy of the region into the buffer *buffername*, at the location of point in that buffer. If there is no buffer with that name, one is created. If you append text into a buffer that has been used for editing, the copied text goes into the middle of the text of the buffer, wherever point happens to be in it.

Point in that buffer is left at the end of the copied text, so successive uses of **C-X A** accumulate the text in the specified buffer in the same order as they were copied. If **C-X A** is given an argument, point in the other buffer is left before the copied text, so successive uses of **C-X A** add text in reverse order.

Strictly speaking, **C-X A** does not always append to the text already in the buffer; but if **C-X A** is the only command used to alter a buffer, it does always append to the existing text because point is always at the end.

You can retrieve the accumulated text from that buffer with **M-X Insert Buffer**; this too takes *buffername* as an argument. It inserts a copy of the text in buffer *buffername* into the selected buffer. You could alternatively select the other buffer for editing, perhaps moving text from it by killing or with **C-X A**. See chapter 19 [Buffers], page 99, for background information on buffers.

Instead of accumulating text within ZMACS, in a buffer, you can append text directly into a file with the command **M-X Append To File**, taking *filename* as an argument. It adds the text of the region to the end of the specified file. **M-X Prepend to File** adds the text to the beginning of the file instead. The file is changed immediately on the file server. These commands are normally used with files that are *not* being visited in ZMACS. Using them on files that ZMACS is visiting can produce confusing results, because the text inside ZMACS for those files will not change.

11.3.2 Copying Text Using Registers

When you want to insert a copy of the same piece of text frequently, it may be impractical to

use the kill history, since each subsequent kill moves the piece of text farther down on the history. It becomes hard to keep track of what argument is needed to retrieve the same text with **C-Y**. An alternative is to store the text in a *register* with **C-X X** (**Put Register**) and then retrieve it with **C-X G** (**Get Register**).

C-X X r Copy region into register *r*.

C-X G r Insert text contents of register *r*.

M-X List Registers

Print list of registers used, and their contents.

C-X X r stores a copy of the text of the region into the register named *r*. *r* is a single graphic character; it is not allowed to use **CONTROL**, **META**, etc. Given a numeric argument, **C-X X** deletes the text as well.

C-X G r inserts in the buffer the text from register *r*. Normally it leaves point before the text and places the mark after, but with a numeric argument it puts point after the text and the mark before.

These registers are the same as used for recording locations, but each register can hold both text and a location. Storing new text does not affect the recorded location, and vice versa. See section 10.4 [Named Marks], page 42.

M-X List Registers prints a list of all registers that you have used, giving for each one the text it contains (or **[EMPTY]** if none) and the location recorded in it, if any. The text is summarized by showing only the beginning or the end, with text in the middle, and any leading or trailing whitespace, replaced by dots. The location is shown by displaying the text surrounding it, with **-|-** in the middle indicating where the location is.

12. Undoing Changes

ZMACS in the MIT LISP Machine system has the most powerful facility known for undoing changes to the text. No change or sequence of changes is too drastic to be undone.

Each ZMACS buffer records all the changes that have been made in it, on its *undo list*. This contains a sequence of *undo entries*, each describing how to undo one change to a contiguous portion of the buffer. Often several commands are recorded in one undo entry, and some commands (such as **Query Replace**) can make many undo entries.

C-Shift-U

Undo one batch of contiguous changes.

C-Shift-R

Redo one batch of undone changes.

M-X Discard Undo Information

Forget the undo list.

M-X Print Modifications

Print all lines changed since file was visited.

The command **C-Shift-U** (**Quick Undo**), when there is no region, undoes the changes recorded by the last undo item and removes that item from the list. It moves point to the beginning of the undone change, ensuring that you can see the effects.

Each buffer has a *redo list* that is much like the undo list. Undoing a change makes an entry on the redo list saying how to undo the undoing. The command **C-Shift-R** (**Quick Redo**) undoes the last entry on the redo list; that is to say, it redoes the last change undone (not counting those already redone). Redoing makes a new entry on the undo list, so that the redone change can be undone again with **C-Shift-U**.

You can think of **C-Shift-U** as taking the text back into the past, and **C-Shift-R** as returning toward the present. If the buffer was "unmodified" before a change was made, it is marked as "unmodified" after that change is undone. When all the recorded changes have been undone, the buffer text is the same as it was when it was first created, or when it was last saved.

It is safest to redo a change just after you undo it. If you undo a change, make other changes, and then try to redo, some of the intervening changes might be lost as the text is returned to the state it was in before the undo. Because of this, **C-Shift-R** asks for confirmation if any other changes (except for other undoing and redoing) have been made since the last undo.

If you notice that a buffer has been modified accidentally, the easiest way to recover is to type **C-Shift-U** repeatedly until it refuses to go any farther. At this time, all the modifications you made have been cancelled. If you do not remember whether you changed the buffer deliberately, type **C-Shift-U** once, and when you see the last change you made undone, you will remember why you made it. Then type **C-Shift-R** to make the change again if you wanted it. Another way to find out what you have done to the buffer in the current session is to type **M-X Print Modifications**, which prints (as typeout) all the lines in the current buffer that have been changed in any way during the current session.

You are not limited to undoing the last change you made. If there is an active region, **C-Shift-U** undoes the last change in that region, even if other subsequent changes have been made outside the region. For example, to undo the last change made in a particular LISP function, go to that function, type **C-M-H** to put the region around it, and then type **C-Shift-U**. **C-Shift-U** leaves the region in effect and does not move point, so it can be repeated to undo several changes in the same region.

C-Shift-U when there is a region can undo only the last change that is entirely within the region. If there is a more recent change that was partly inside and partly outside the region, then you cannot undo anything in this region. If the two changes do not actually overlap, then by moving point or the mark you can find a smaller region that contains the change you wish to undo and does not overlap the other change. Then **C-Shift-U** will work. If the two changes do overlap, you cannot undo the earlier one without undoing the later one first. This is a good thing, because overlapping changes are not independent, and undoing them in the wrong order would produce incorrect results.

Each entry in the undo list or the redo list identifies the kind of editing operation that made the changes it records. For example, the **M-Q** command (**Fill Paragraph**) always makes its own undo entry, which says it is for a "Fill". Undoing this entry prints **'Fill undone'**. Yanking always makes a separate undo entry, but yank-pop operations (**M-Y**) are grouped together with the original yank.

Most ZMACS commands do not create their own undo-list entries. These commands are classified as "small changes" and are grouped into "small change" entries. These changes are grouped in order to increase the amount of distance you cover with each **C-Shift-U** command. If the variable **zwei:undo-save-small-changes*** is set to **nil**, small change commands are not recorded for undoing. Now that the undo facility works reliably, there is little reason to turn off such saving.

A small change command adds to an unfinished small change undo-list entry if the new change is adjacent to the change already recorded by the entry. For example, inserting two characters next to each other in the text is undone as one operation, even if other commands that do not

modify the text intervene in time. If a small change command cannot add to an existing entry, it starts a new small change entry. This always happens if the last existing entry is not a small change entry.

Each undo entry or redo entry uses a couple of relocating buffer pointers. Editing on a line that has many relocating buffer pointers is slowed down by the need to relocate them. As a result, a long editing session can build up enough undo entries to cause noticeable delays. For this reason, it is sometimes useful to forget the undo list and give up the ability to undo any changes made so far. This can be done explicitly with the command **M-X Discard Undo Information**. It also happens automatically when you save a buffer. This automatic action can be disabled by setting the variable `zwei:*discard-undo-info-on-saving*` to `nil`. Then undo information is never discarded except when you request it explicitly.



13. The Various Quantities Command

ZMACS recognizes many kinds of syntactic units (characters, words, lines, pages, etc.) and knows several operations to perform on those units (moving, killing, marking, case changing, etc.) Most ZMACS commands are single characters that specify at once the type of syntactic unit to operate on and what to do with it; for example, **Control-F** specifies forward motion over units of characters. These commands are quick in use but are a lot to remember. An alternative that is easier to remember is the **C-Shift-X** command (**Various Quantities**), which allows you to specify the operation with one character and the syntactic unit with another. Using **C-Shift-X** if you remember ten subcommands for operations such as forward or backward motion or killing, and ten subcommands for textual units such as characters, words and lines, you can do a hundred different things.

The first character you type after **C-Shift-X** should specify the operation to perform. Then you should type another character specifying the textual unit to operate on. For example, **C-3 C-Shift-X U S** converts three sentences after point to upper case, since **U** means "convert to upper case" and **S** means "sentences".

Here is a table of the allowed operation subcommands.

F	Move forward.
B	Move backward.
K	Kill forward.
R	Rub out (kill backward).
X	Exchange.
Q	Set mark beyond.
U	Convert to upper case.
L	Convert to lower case.
S	Save on kill history.
C	Copy (save, then yank a second copy).
Z	Reverse.

Here are the allowed textual unit subcommands.

C	Characters.
L	Lines.
W	Words (see section 24.2 [Words], page 126).

- S** Sentences (see section 24.3 [Sentences], page 127).
 - P** Paragraphs (see section 24.4 [Paragraphs], page 128).
 - S-expressions (see section 25.2 [Lists], page 136).
 - (**
 -)** Lists (see section 25.2 [Lists], page 136).
 - A** Atoms. Like operating on s-expressions except that parentheses are ignored, so only the atoms are counted. This is like what the word commands do in Atom Word mode (see section 31.1 [Minor Modes], page 191), but this interface is available regardless of mode.
 - D** Defuns (see section 25.3 [Defuns], page 138).
- CLEAR-SCREEN** Pages (see section 24.5 [Pages], page 129). Note that **CLEAR-SCREEN** is the same character as **PAGE**.
- H** The whole buffer.

A numeric argument to **C-Shift-X** says how many of the specified textual unit, before or after point, should be operated on. A negative argument reverses the direction of scan, away from point. This is necessary if you wish to operate on text before point with an operation other than moving point or killing.

The “reverse” operation is interesting only if given a numeric argument larger than one (or less than -1). It reverses the ordering of the textual objects it is operating on. For example, **C-6 C-Shift-X Z W** reverses the six words following point.

The “copy” operation, when given a numeric argument n , finds n of the specified textual unit and copies them all as one block. It makes only one copy.

The “exchange” operation treats its numeric argument like the exchange commands. See section 16.2 [Transposition], page 72.

Numeric arguments are meaningless with **H** for “whole buffer”, and certain operations such as “exchange” are also meaningless with it.

14. Controlling the Display

Since only part of a large buffer fits in the window, ZMACS tries to show the part that is likely to be interesting. The display control commands allow you to ask to see a different part of the text. This is also known as *scrolling*.

If a buffer contains text that is too large to fit entirely within a window that is displaying the buffer, ZMACS shows a contiguous section of the text. The section shown always contains point. As you change the text, ZMACS always tries to keep the same position in the text at the top of the window. A new position moves to the top of the window only if this is necessary to keep point visible, or if you request it explicitly with a display control command.

C-L Clear window and redisplay, putting point at a specified vertical position.

(CLEAR-SCREEN)

Clear entire screen and redisplay all windows.

C-V Scroll forwards (a windowful or a few lines).

M-V Scroll backwards.

M-R Move point to the text at a given vertical position within the window.

C-M-R Shift the LISP function in which point is located onto the screen.

The basic display control command is **C-L (Recenter Window)**. In its simplest form, with no argument, it clears the entire selected window and then redisplay it after scrolling the text so that point is centered vertically.

C-L with a positive argument repositions text to put point the specified number of lines down from the top. An argument of zero puts point on the very top line. Point does not move with respect to the text; rather, the text and point move rigidly on the screen. **C-L** with a negative argument puts point that many lines from the bottom of the window. For example, **C-U - 1 C-L** puts point on the bottom line, and **C-U - 5 C-L** puts it five lines from the bottom. **C-L** with an argument does not clear the window; it moves the bits on the screen as appropriate to speed up redisplay.

(CLEAR-SCREEN) (Complete Redisplay) is a more drastic way to clear everything and redisplay. It the entire screen and redisplay every window on it—not just ZMACS. It does not change the position of the text in the window, however.

The *scrolling* commands **C-V** and **M-V** let you move the whole display up or down a few lines. **C-V (Next Screen)** with an argument shows you that many more lines at the bottom of the window,

another **C-S** to move to the next occurrence of the search string. This can be done any number of times. If you overshoot, you can cancel some **C-S**'s with **(RUBOUT)**.

After you exit a search, you can search for the same string again by typing just **C-S C-S**: one **C-S** command to start the search and then another **C-S** to mean "search again".

If your string is not found at all, the echo area says 'Failing I-Search'. The cursor is after the place where ZMACS found as much of your string as it could. Thus, if you search for 'FOOT', and there is no 'FOOT', you might see the cursor after the 'FOO' in 'FOOL'. At this point there are several things you can do. If your string was mistyped, you can rub some of it out and correct it. If you like the place you have found, you can type **(ALTMODE)** or some other ZMACS command to "accept what the search offered". Or you can type **C-G**, which removes from the search string the characters that could not be found (the 'T' in 'FOOT'), leaving those that were found (the 'FOO' in 'FOOT'). A second **C-G** at that point cancels the search entirely, returning point to where it was when the search started.

The **C-G** "quit" command does special things during searches; just what it does depends on the status of the search. If the search has found what you specified and is waiting for input, **C-G** cancels the entire search. The cursor moves back to where you started the search. If **C-G** is typed when there are characters in the search string that have not been found—because ZMACS is still searching for them, or because it has failed to find them—then the search string characters that have not been found are discarded from the search string. With them gone, the search is now successful and waiting for more input, so a second **C-G** will cancel the entire search.

To search for a special character such as **(BREAK)** or **(ABORT)**, you must quote it by typing **C-Q** first. This function of **C-Q** is analogous to its meaning as a ZMACS command: it causes the following character to be treated the way a graphic character would normally be treated in the same context.

The **(ABORT)** character at any time during an incremental search cancels the entire search instantly and moves point back to where the search started.

You can change to searching backwards with **C-R**. If a search fails because the place you started was too late in the file, you should do this. Repeated **C-R**'s keep looking for more occurrences backwards. A **C-S** starts going forwards again. **C-R**'s can be rubbed out just like anything else. If you know that you want to search backwards, you can use **C-R** instead of **C-S** to start the search, because **C-R** is also a command (**Reverse Incremental Search**) to search backward.

15.1 Searching and Case

All sorts of searches in ZMACS normally ignore the case of the text they are searching through; if you specify searching for 'FOO', then 'Foo' and 'foo' are also considered a match. If you do not want this feature, set the variable `zwei:*alphabetic-case-affects-search*` to nil. See section 31.2 [Variables], page 192.

15.2 Nonincremental Search

ZMACS also has a conventional nonincremental search command, which requires you to type the entire search string before searching begins. To get this command, type **C-S** `ALTMODE`.

The basic way to use nonincremental search is to type a string to search for, followed by `ALTMODE`. At this time, the search is done, and point moves if the search string is found. If the string is not found, point does not move, but the search is terminated anyway. `ALTMODE` exits nonincremental search whether or not a string is found.

Control characters have special meanings while the nonincremental search argument is being read. Only **C-S** and **C-R** cause any immediate action. The others just change the details of what will be done when searching takes place.

`ALTMODE`

- `END` Search right away, move point if something is found, then unconditionally exit from searching.
- C-B** When the search is done, search forward from the beginning of the buffer.
- C-D** Take the string to search for from the history of recent search strings.
- C-E** When the search is done, search backward from the end of the buffer.
- C-F** When the search is done, if the location found is off the screen, reposition the window with point at the top (rather than centered as usual).
- C-G** Quit the search command; return to top level. No search will be done.
- C-L** Redisplay the screen. This has no effect on what kind of search will be done or on what string will be searched for.
- C-Q** Quote the next character; include it in the search string even if it is special.
- C-R** Reverse the direction in which the searching will be done.
- C-S** Search right away for the search string specified so far; then remain inside nonincremental search, so that more searching can be done.

CLEAR-INPUT

- C-U** Cancel all the characters specified so far for the search string.
- C-V** When searching is done, accept only matches surrounded by word delimiter characters.
- C-W** When searching is done, do word search (see below).
- C-Y** Append the previous search string, from the search string history, to the currently specified search string.

RUBOUT

Cancel the last character of the search string.

Some of the special characters affect the kind of searching to be done, others (including graphic characters) change the string to be searched for, while others (**C-S** and **ALTMODE**) search for the search string already specified, in the manner already specified.

All kinds of searches make entries on the search string history, and incremental search can use the latest entry on the history (when you type **C-S C-S** to search for the previous search string), but only nonincremental search allows you to access previous entries on the search string history. You do this using **C-D** repeatedly. Each **C-D** gets the previous entry on the history. This is not annoying to do, since no searching happens until you type **C-S** or **ALTMODE**.

C-S ALTMODE starts out by invoking incremental search, which is specially programmed to invoke nonincremental search if the argument you give it is empty. (Such an empty argument would otherwise be useless.) However, nonincremental search is a ZMACS command function in its own right (**String Search**) and you could connect it directly to a command key. The command function **Reverse String Search** also exists. It searches backward in the buffer unless directed otherwise.

15.3 Word Search

Word search searches for a sequence of words without regard to how the words are separated. More precisely, you type a string of many words, using single spaces to separate them, and the string can be found even if there are multiple spaces or newlines between the words. Other punctuation such as commas or periods must match exactly. This is useful in conjunction with documents formatted by text formatters. If you edit while looking at the printed, formatted version, you can't tell where the line breaks are in the source file. With word search, you can search without having to know this.

Word search is a special case of nonincremental search and is invoked with **C-S ALTMODE C-W**. This is followed by the search string, which must always be terminated with **ALTMODE** or **END**. Being nonincremental, this search does not start until the argument is terminated.

You do not even have to type each word in full, in a word search. An abbreviation is good enough. Word search finds the first occurrence of a sequence of words whose beginnings match the words of the argument.

15.4 LISP Pattern Search

The command **C-Shift-S** (**LISP Match Search**) searches for a list that matches a pattern. The pattern is a fragment of LISP code, which may contain parentheses that need not be balanced. Most characters in the pattern must match exactly, though differences in whitespace are allowed in places where it has no effect on the meaning of LISP code. Two pattern elements are special: ****** matches any s-expression, and **'...'** matches any sequence of s-expressions.

For example, to find the next **if** that has an else-clause, use

```
(if ** ** ** ...)
```

as the argument to **C-Shift-S**. It matches a list whose first element is **if** and which has at least three more elements.

A negative numeric argument to **C-Shift-S** specifies searching backwards in the buffer. An empty pattern argument means to use the previous pattern.

15.5 Replacement Commands

Global search-and-replace operations are not needed as often in ZMACS as they are in other editors, but they are available. In addition to the simple **Replace String** command function, similar to that found in most editors, there is a **Query Replace** command function, which asks you, for each occurrence of the pattern, whether to replace it.

15.5.1 Replace String

C-% Replace every occurrence of *string* with *otherstring*.

To replace every instance of **'foo'** after point with **'bar'**, use the command **C-%** (**Replace String**) with the two arguments **'foo'** and **'bar'**. Replacement occurs only after point, so if you want to

cover the whole buffer you must go to the beginning first. All occurrences up to the end of the buffer are replaced, except that if a numeric argument is given, only that many occurrences are replaced. The number of replacements done is printed when the command finishes.

If the arguments to **Replace String** are in lower case, it preserves case when it makes a replacement. Thus, a lower case 'foo' is replaced by a lower case 'bar', 'FOO' is replaced by 'BAR', and 'Foo' by 'Bar'. If upper case letters are used in the second argument, they remain upper case every time that argument is inserted. If upper case letters are used in the first argument, the second argument is always substituted exactly as given, with no case conversion. Likewise, if the variable `zwei:*case-replace-p*` is set to `nil`, replacement is done without case conversion. If `zwei:*alphabetic-case-affects-search*` is set to `nil`, case is significant in matching occurrences of 'foo' to replace.

15.5.2 Query Replace

M-%

M-X Query Replace

Replace some occurrences of one string with another string.

M-X Query Replace Last Kill

Replace some occurrences of last killed string with current region contents.

M-X Query Exchange

Replace each of two string with the other.

M-X Multiple Query Replace

Replace each of several strings with its own replacement string.

M-X Multiple Query Replace From Buffer

Similar, taking arguments from a ZMACS buffer.

If you want to change only some of the occurrences of 'foo', not all, then you cannot use an ordinary **Replace String**. Instead, use **M-% (Query Replace)**. This command finds occurrences of 'foo' one by one, displays each occurrence and asks you whether to replace it. A numeric argument to **Query Replace** tells it to consider only occurrences of 'foo' that are bounded by word-delimiter characters.

The things you can type when you are shown an occurrence of 'foo' are:

- (SPACE)** to replace the 'foo' with 'bar'. This preserves case, just like **Replace String**, provided `zwei:*case-replace*` is non-`nil`, as it normally is.
- (RUBOUT)** to skip to the next 'foo' without replacing this one.

- . to replace this 'foo' and display the result. You are then asked for another input character, except that since the replacement has already been made, **(RUBOUT)** and **(SPACE)** are equivalent.
- (ALTMODE)** to exit without doing any more replacements.
- . to replace this 'foo' and then exit.
- ! to replace all remaining occurrences of 'foo' without asking again.
- to go back to the location of the previous 'foo' (or what used to be a 'foo'), in case changed it by mistake. This works by popping the point pdl, possible because **Query Replace** pushes point onto the point pdl each time an occurrence of 'foo' is found.
- C-R** to enter a recursive editing level, in case the 'foo' needs to be edited rather than just replaced with a 'bar'. When you are done, exit the recursive editing level with **(END)** and the next 'foo' will be displayed. See section 29.1 [Recursive Edit], page 179.
- C-W** to delete the 'foo', and then start editing the buffer. When you are finished editing whatever is to replace the 'foo', exit the recursive editing level with **(END)** and the next 'foo' will be displayed.
- C-L** to redisplay the screen and then give another answer.

If you type any other character, the **Query Replace** is exited, and the character executed as a command. To restart the **Query Replace**, use **C-X (ALTMODE)**, which repeats the **Query Replace** because that used the minibuffer to read its arguments. See section 7.2 [Repetition], page 33.

Often you may decide to replace one string with another after making the change in one place. In such a case, **M-X Query Replace Last Kill** may be useful. It replaces interactively like **Query Replace** but does not use the minibuffer to get the strings to search for and replace with. The last killed string is used as the string to search for, and the current contents of the region when the command is given become the string to replace it with.

A useful related command is **M-X Multiple Query Replace**, which reads any number of pairs of arguments terminated by an empty argument, and takes each pair of arguments as a string to replace followed by what to replace it with. The buffer is searched repeatedly for the next occurrence of any of the strings to replace; if the user says to replace it, it is replaced with its corresponding replacement string. A special case of this is **M-X Query Exchange**, which reads two arguments, searches for both, and replaces each with the other.

M-X Multiple Query Replace From Buffer is another way to do a multiple query replace. It does the same work as **M-X Multiple Query Replace** but reads its arguments differently: it reads the name of a ZMACS buffer that should contain text describing the replacements to be made. The buffer should contain two LISP strings on each line, with a space in between. The first string is a string to search for, and the second is what to replace it with. Here is an example of text


specifying three replacements:

```
"foo" "bar"  
"old" "new"  
"obsolete" "current"
```

15.6 Other Search-and-Loop Commands

Here are some other command functions that find occurrences of a string. They all operate from point to the end of the buffer.

M-X List Matching Lines

Print each line that follows point and contains a match for the specified string. A numeric argument specifies the number of context lines to print before and after each matching line; the default is none. The lines of output are mouse-sensitive; click  on one of them to move point to the text line described by that line of output.

M-X Count Occurrences

Print the number of matches following point for the specified string.

M-X Delete Non-Matching Lines

Delete each line that follows point and does not contain a match for the specified string.

M-X Delete Matching Lines

Delete each line that follows point and contains a match for the specified string.

15.7 Extended Search Characters

Many commands that search for substrings are not limited to exact match. When giving the argument, you can specify *extended search characters*, which give the effect of regular expressions. Regular expression operators are specified by typing two-character sequences that start with **Control-H**. (**C-H** has this definition only in the minibuffer and only when extended search characters are allowed.) All ordinary characters in the argument stand for themselves, requiring exact match.

The minibuffer prompt line says '(Extended search characters)' at the far right whenever extended search characters are allowed as input.

Here are the allowed C-H sequences:

C-H &

C-H ^

C-H C-A Any of these inserts the infix operator character **(AND)**. A line must contain a match for each of 'foo' and 'bar' in order to match 'foo**(AND)**bar'. Thus, 'to**(AND)**of' matches any line that contains 'to' and also contains 'of'.

C-H v

C-H C-O Either of these inserts the infix operator character **(OR)**. A match for 'foo**(OR)**bar' is anything that matches either 'foo' or 'bar'.

C-H ~

C-H -

C-H C-N Any of these inserts the prefix operator character **(NOT)**. A match for '**(NOT)**b' is anything that does not match 'b'. **(NOT)** has tight binding, so it applies only to the following character, unless it is followed by an expression bracketed with **(OPEN)** and **(CLOSE)**.

C-H **(SPACE)**

This inserts the extended search character **(WHITESPACE)**, which matches any whitespace character.

C-H A

This inserts the extended search character **(ALPHABETIC)**, which matches any alphabetic character.

C-H -

This inserts the extended search character **(DELIMITER)**, which matches any delimiter character.

C-H C-X

This inserts the extended search character **(ANY)**, which matches any character.

C-H *

This inserts the prefix operator character **(SOME)**. '**(SOME)**b' matches any sequence of zero or more matches for 'b'. **(SOME)** has tight binding, so it applies only to the following character, unless it is followed by an expression bracketed with **(OPEN)** and **(CLOSE)**.

C-H (

C-H)

These insert the extended search grouping characters **(OPEN)** and **(CLOSE)**. An extended search pattern surrounded by **(OPEN)** and **(CLOSE)** matches the same thing it would match without them, but it can serve as a unit as an argument to a preceding **(SOME)** or **(NOT)**.



16. Commands for Fixing Typos

In this chapter we describe the commands that are especially useful for the times when you catch a mistake in your text just after you have made it, or change your mind while composing text on line.

- (RUBOUT)** Delete last character.
- M-(RUBOUT)** Kill last word.
- C-X (RUBOUT)** Kill to beginning of sentence.
- C-T** Transpose two characters.
- C-X C-T** Transpose two lines.
- C-X T** Transpose two arbitrary regions.
- M-- M-L** Convert last word to lower case. **Meta--** is Meta-minus!
- M-- M-U** Convert last word to all upper case.
- M-- M-C** Convert last word to lower case with capital initial.
- M-*** Fix up omitted shift key on digit.
- M-§** Check and correct spelling of word.

16.1 Killing Your Mistakes

The **(RUBOUT)** command is the most important correction command. When used among graphic (self-inserting) characters, it can be thought of as canceling the last character typed.

When your mistake is longer than a couple of characters, it might be more convenient to use **M-(RUBOUT)** or **C-X (RUBOUT)**. **M-(RUBOUT)** kills back to the start of the last word, and **C-X (RUBOUT)** kills back to the start of the last sentence. **C-X (RUBOUT)** is particularly useful when you are thinking of what to write as you type it, in case you change your mind about phrasing. **M-(RUBOUT)** and **C-X (RUBOUT)** save the killed text for **C-Y** and **M-Y** to retrieve. See section 11.2 [Yanking], page 47.

M-(RUBOUT) is often useful even when you have typed only a few characters wrong, if you know you are confused in your typing and aren't sure exactly what you typed. At such a time, you cannot correct with **(RUBOUT)** except by looking at the screen to see what you did. It requires less thought to kill the whole word and start over again.

16.2 Transposing Text

The common error of transposing two characters can be fixed, when they are adjacent, with the **C-T** command (**Exchange Characters**). Normally, **C-T** transposes the two characters on either side of point. When given at the end of a line, rather than transposing the last character of the line with the newline, which would be useless, **C-T** transposes the last two characters on the line. So, if you catch your transposition error right away, you can fix it with just a **C-T**. If you don't catch it so fast, you must move the cursor back to between the two transposed characters. If you transposed a space with the last character of the word before it, the word motion commands are a good way of getting there. Otherwise, a reverse search (**C-R**) is often the best way. See chapter 15 [Search], page 61.

To transpose two lines, use the **C-X C-T** command (**Exchange Lines**). **M-T** (**Exchange Words**) transposes words and **C-M-T** (**Exchange Sexps**) transposes s-expressions.

A more general transpose command is **C-X T** (**Exchange Regions**). This transposes two arbitrary blocks of text, which need not even be next to each other. To use it, set the mark at one end of one block, then at the other end of the block; then go to the other block and set the mark at one end, and put point at the other. In other words, point and the last three locations on the point pdl delimit the two blocks. It does not matter which of the four locations point is at, or which order the others were marked. **C-X T** transposes the two blocks of text thus identified, and relocates point and the three marks without changing their order.

16.3 Case Conversion

A very common error is to type words in the wrong case. Because of this, the word case-conversion commands **M-L**, **M-U** and **M-C** have a special feature when used with a negative argument: they do not move the cursor. As soon as you see you have mistyped the last word, you can simply case-convert it and go on typing. See section 23.1 [Case], page 119.

Another common error is to type a special character and miss the shift key, producing a digit instead. There is a special command for fixing this: **M-*** (**Uppcase Digit**), which fixes the last digit before point in this way (but only if that digit appears on the current line or the previous line. Otherwise, to minimize random effects of accidental use, **M-*** does nothing). Once again, the cursor does not move, so you can use **M-*** when you notice the error and immediately continue typing. This command is most useful for changing '8' to '*'.

16.4 Checking and Correcting Spelling

To check the spelling of the word before point, and optionally correct it as well, use the command **M- $\$$** (**Correct Word Spelling**). This command sends the word to a SPELL server for correction. The SPELL server is a program running on another machine, contacted via the Chaos net.

If the SPELL server recognizes the word as a correctly spelled one (although not necessarily the one you meant!), **M- $\$$** prints **'Found it'** in the echo area, or possibly **'Found it because of root'** if the specified word is recognized as a derivative of *root*.

If the server cannot at all recognize the word, **M- $\$$** prints **'Couldn't find it'**.

If the server finds that the word is not correct but is close to some correctly spelled words, **M- $\$$** gives you the option of choosing one of them as a replacement. The list of suggested replacements is printed on top of the current window, as typeout. You should then type one of these characters:

- (SPACE)** Exit and make no changes.
- R** Replace the word with a new word that you will enter via the minibuffer.
- A, B, ...** Replace the word with the first, second, etc. suggested possibility in the list.

If you elect to replace the word, you are asked whether to do a **Query Replace** to replace any or all occurrences of the misspelling throughout the buffer. Answer **Y** or **N**. See section 15.5 [**Query Replace**], page 65.

When you use **M- $\$$** , point need not be immediately after the word you want to correct; it can be in the middle, or following any word-separator characters after the end of the word. Note that the major mode you are using affects which characters are word separators. See section 31.5 [**Syntax Table**], page 201.

It may be that no machine accessible at your site provides a spell server, or that all the machines that have them are down. In this case, **M- $\$$** just prints a message in the echo area to inform you of this fact.



17. File Handling

The basic unit of stored data is the file. Each program, each paper, lives usually in its own file. To edit a program or paper, the editor must be told to examine the file and prepare a buffer containing a copy of the file's text. This is called *visiting* the file. Editing commands apply directly to text in the buffer; that is, to the copy inside ZMACS. Your changes only appear in the file itself when you save the buffer back into the file.

In addition to visiting and saving files, ZMACS can delete, copy, rename, and append to files, and operate on file directories.

17.1 File Names

Most ZMACS commands that operate on a file require you to specify the file name. (Saving and reverting are exceptions; the buffer knows which file name to use for them.) File names are specified using the minibuffer. There is always a *default file name* that will be used if you type just **RETURN**, entering an empty argument. Normally the default file name is the name of the file visited in the current buffer; this makes it easy to operate on that file with any of the ZMACS file commands. If the current buffer is a Dired buffer, the default file name is the file on the current line.

File names are made up of six components: the *host*, *device*, *directory*, *name*, *type* and *version*. Here are some examples, for various file server types:

```
OZ:KANSAS:<L.SYS>QFCTNS.LISP
```

Here **OZ** is the host (a Twenex host), **KANSAS** the device, **<L.SYS>** the directory, **QFCTNS** the name, and **LISP** the type. (The type component is conventionally used to indicate what kind of data the file holds; this file is a LISP program.) The version is not specified explicitly, which means that the latest version will be used.

```
PREP:/u/rms/gnu/gnu.tasks
```

Here **PREP** is the host (a Unix host), there is no device name as Unix does not support them, **/u/rms/gnu/** is the directory, **gnu** is the name, and **tasks** is the type. Unix does not support versions, and there is only one version of any file.

When you specify a file name, you can specify it completely by giving all the components. (The version would still be omitted, usually, since normally you want to look at the latest version and save new versions.) You can also omit some of the components; then they are taken from the default file name. For example, if the default is **OZ:KANSAS<L.SYS>QFCTNS.LISP** and you want to visit **OZ:KANSAS<L.SYS>EVAL.LISP**, it is sufficient to type **C-X C-F EVAL** **RETURN**; the other components you want come from the defaults.

The default file name is shown in the mode line while a file name is being read, so that you can see which components it is necessary for you to specify and which ones will default to the values you want.

Note that if you specify the name component but not the type component, ZMACS uses the conventional type for LISP programs on the host you are using, rather than the type component from the default file names. Also, if you specify the name but not the version, the default for the version is always "newest version".

See the chapter on pathnames in the *LISP Machine Manual* for more information on pathname components and their defaulting.

Another way you can use the default file name with some modification is to yank it into the minibuffer and edit it into the name you want. To yank it, use **C-Shift-Y** (**Yank Default String**). Edit with ordinary ZMACS commands and then submit the result with **RETURN**.

ZMACS keeps a history of the file names you have specified previously. To yank from this list, type **C-M-Y** (**Yank Previous Input**). Numeric arguments can be used to refer to earlier entries in the history, as with **C-Y**, and **M-Y** can be used to move around in the history. An argument of zero causes a list of the recent elements of the history to be printed. See section 11.2 [Yanking], page 47. **C-M-Y** is defined this way only in the minibuffer.

Completion is another tool that makes it easier to specify long file names. It allows you to use an abbreviation that specifies uniquely one of the existing files. See section 7.1 [Completion], page 30.

17.2 Visiting Files

C-X C-F Visit a file.

C-X C-V Visit a different file instead of the one visited last.

M-X Find File No Sectionize

Visit a file but do not sectionize it (scan it for function definitions).

M-X Find File Background

Visit a file, but meanwhile continue editing.

M-X Find System Files

Visit all the files that make up a system.

Visiting a file means copying its contents into ZMACS where you can edit them. ZMACS makes a new buffer for each file that you visit. We say that the buffer is *associated* with the file that it was made to hold. ZMACS constructs the buffer name from the file name by permuting its elements so that the 'name' and 'type' components come first. For example, a file named `goo:/usr/rms/ZMACS.tex` would get a buffer named `'ZMACS.tex /usr/rms goo:'`.

Since the current buffer name always appears in the mode line, you can tell instantly from it which file you are editing. If the file is on a file system that supports multiple versions, the current version number (of the version that ZMACS last read or saved) appears inside parentheses after the buffer name.

The changes you make with ZMACS are made in the ZMACS buffer. They do not take effect in the file that you visited, or any place that lasts past the current LISP Machine session, until you save the buffer. Saving the buffer means that ZMACS writes the current contents of the buffer into its associated file. See section 17.3 [Saving], page 79.

If a buffer contains changes that have not been saved, the buffer is said to be *modified*. This is important because it implies that some changes will be lost if the buffer is not saved. The mode line displays a star before the buffer name if the current buffer is modified.

To visit a file, use the command **C-X C-F (Find File)**. Follow the command with the name of the file you wish to visit, terminated by a **(RETURN)**. Any file name components that you do not specify are taken from the *default file name*, which is displayed in the mode line while you are entering the file name argument, so that you can tell which components you can safely omit.

The file name to find is a standard minibuffer argument (see chapter 7 [Minibuffer], page 29) and completion is available with **(ALTMODE)** and **(END)** (see section 7.1 [Completion], page 30). While in the minibuffer, you can abort the **C-X C-F** command by typing **(ABORT)**.

If the **C-X C-F** command completes successfully, it prints a message in the echo area giving the truename and size of the file visited. By this time, it has created a new buffer containing a copy of the text of the visited file, and selected that buffer for editing.

If you visit a file that is already in ZMACS, it does not make another copy. It selects the existing buffer containing that file. However, before doing so, it checks that the file itself has not changed since you visited or saved it last. If the file has changed, a warning message is printed. See section 17.3.1 [Simultaneous Editing], page 80.

What if you want to create a file? Just visit it. ZMACS prints '(New File)' in the echo area, but in other respects behaves as if you had visited an existing empty file. If you make any changes and save them, the file is created. But if the variable `zwei:*find-file-not-found-is-an-error*` is non-nil, visiting a nonexistent file is an error. Then the only way to create a file is to make a buffer with `C-X B` and then save it in the desired file. It is hard to imagine why anyone would want this inconvenience, but apparently some people do.

If you visit a nonexistent file unintentionally (because you typed the wrong file name), use the `C-X C-V (Find Alternate File)` command to visit the file you wanted. `C-X C-V` is similar to `C-X C-F`, but it kills the current buffer, and is allowed only after you have visited a nonexistent file and made no changes in it.

Normally ZMACS *sectionizes* a file when it is read in. This means scanning the lines of the file to find the beginning of each LISP function, or each section of a manual, and so on. In some cases, such as when you are examining an old version of a file and want ZMACS to use the latest version to find any function defined in the file, you may wish to prevent sectionization. To do this, use the command `M-X Find File No Sectionize` instead of `C-X C-F` to visit the file. See section 26.1 [Sectionization], page 153.

The command `M-X Find File Background` visits the specified file like `C-X C-F`, but does not wait for visiting to be finished. Instead, you can continue to edit the file you were editing, while the new file is visited by a background process. A notification is printed when the file is fully visited, but your editing is not interrupted. From that time on, you can select its buffer with `C-X B`.

If the variable `zwei:*find-file-early-select*` is non-nil, the buffer created for a newly visited file is selected and displayed as soon as enough of the file has been read to fill the window. But you still cannot begin editing until the entire file has been read.

17.2.1 Visiting Multiple Files

There are two commands in ZMACS for visiting a group of files all at once: specify a file name pattern containing wild cards in `C-X C-F`, use `M-X Find System Files`. You can also do this with `Dired`; See chapter 18 [Dired], page 91.

If the argument to **C-X C-F** uses wildcards, it finds the list of all file names that match the pattern and visits the files one by one.

M-X Find System Files reads the name of a *system*—a collection of files that make up one program—and visits each of the source files in the system. Systems are declared using **defsystem**; see the chapter “Maintaining Large Systems” in the *LISP Machine Manual* for information on them.

17.3 Saving Files

C-X C-S Save the current buffer in its associated file.

M- Forget that the current buffer has been changed.

C-X C-W Save the current buffer in a specified file, and associate them.

M-X Save All Files

Save any or all buffers in their associated files.

M-X Set Visited File Name

Associate the current buffer with a file.

When you wish to save the file and make your changes permanent, type **C-X C-S (Save File)**. After saving is finished, **C-X C-S** prints a message such as

Written: PREP:/u/rms/.emacs -- 431 characters

giving the truename and size of the file written. If the selected buffer is not modified (no changes have been made in it since the buffer was created or last saved), saving is not really done, because it would be redundant. Instead, **C-X C-S** prints a message in the echo area saying

(No changes need to be written.)

The command **M-X Save All Files** can save any or all modified buffers. First it asks, for each modified buffer, whether to save it. These questions appear as typeout, overlying the buffer text, and should be answered with Y or N. After all questions have been asked, the buffers you have approved are all saved.

If you have changed a buffer and do not want the changes to be saved, you should take some action to prevent it. Otherwise, each time you use **M-X Save All Files** you are liable to save it

by mistake. One thing you can do is type **M-~ (Not Modified)**, which clears out the indication that the buffer is modified. If you do this, none of the save commands will believe that the buffer needs to be saved. (If we take '~' to mean 'not', then **Meta-~** is 'not', metafied.) You could also use **Set Visited File Name** (below) to associate the buffer with a different file name, one that is not in use for anything important. Alternatively, you can undo all the changes made since the file was visited or saved, by reading the text from the file again. This is called *reverting*. See section 17.4 [Reverting], page 81. You could also undo all the changes by repeating the undo command **C-Shift-U** until it refuses to go any farther.

M-X Set Visited File Name alters the name of the file that the current buffer is visiting. It reads the new file name using the minibuffer. It can be used on a buffer that is not visiting a file, too. The buffer's name is changed to correspond to the file it is now visiting in the usual fashion (unless the new name is in use already for some other buffer; in that case, the buffer name is not changed). **Set Visited File Name** does not save the buffer in the newly associated file; it just alters the records inside ZMACS so that, if you save the buffer, it will be saved in that file. It also marks the buffer as "modified" so that **C-X C-S** will save.

If you wish to associate the buffer with a different file and save it right away, use **C-X C-W (Write File)**. It is precisely equivalent to **Set Visited File Name** followed by **C-X C-S**. **C-X C-S** used on a buffer that is not associated with a file has the same effect as **C-X C-W**; that is, it reads a file name, associates the buffer with that file, and saves it there. The default file name in a buffer that is not visiting a file is made by combining the buffer name (as the name component of the pathname) with the other components taken from the last file-visiting buffer that was current.

If ZMACS is about to save a file and sees that the date of the latest version on disk does not match what ZMACS last read or wrote, ZMACS notifies you of this fact, because it probably indicates a problem caused by simultaneous editing and you must correct it.

17.3.1 Protection against Simultaneous Editing

Simultaneous editing occurs when two users visit the same file, both make changes, and then both save them. If nobody were informed that this was happening, whichever user saved first would later find that his changes were lost. ZMACS cannot prevent users from editing simultaneously, but it always warns at least one of the users (the one who saves last) that he is about to lose. If he takes the proper corrective action at this point, he can prevent a problem.

Every time ZMACS saves a buffer, it first checks the last-modification-date of the existing file on disk to see that it has not changed since the file was last visited or saved. If the date does not

match, it implies that changes were made in the file in some other way, and these changes are about to be lost if ZMACS actually does save. To prevent this, ZMACS prints a warning message and asks for confirmation before saving. Occasionally you will know why the file was changed and know that it does not matter; then you can answer 'yes' and proceed. Otherwise, you should cancel the save with **(ABORT)** and investigate the situation.

The first thing you should do when notified of simultaneous editing is list the directory with **C-X C-D** (see section 17.5 [Directory Listing], page 81). This will show the file's current author. You should attempt to contact him to warn him not to continue editing. Often the next step is to compare the file against the text in your buffer with **M-X Source Compare Changes**. **M-X Source Compare Merge** may be useful in preparing a version of the file containing both your changes and the changes newly installed in the file. See section 17.7 [Source Compare], page 84.

Simultaneous editing checks are also made when you visit with **C-X C-F** a file that is already visited. This is not strictly necessary, but it can cause you to find out about the problem earlier, when perhaps correction takes less work.

17.4 Reverting a Buffer

If you have made extensive changes to a file and then change your mind about them, you can get rid of them by reading in the previous version of the file. To do this, use **M-X Revert Buffer**. You must specify the name of the buffer to revert, as an argument, using the minibuffer. Typing just **(RETURN)** specifies the current buffer. Having to type this additional **(RETURN)** acts as a kind of confirmation.

M-X Revert Buffer keeps point at the same distance (measured in lines) from the beginning of the file. If the file was edited only slightly, you will be at approximately the same piece of text after reverting as before. If you have made drastic changes, the same value of point in the old file may address a totally different piece of text.

A reverted buffer is marked "not modified" until another change is made.

Some kinds of buffers whose contents reflect data bases other than files, such as Dired buffers, can also be reverted. For them, reverting means recalculating their contents from the appropriate data base. Buffers created randomly with **C-X B** cannot be reverted; **M-X Revert Buffer** reports an error when asked to do so.

17.5 Listing a File Directory

File servers classify files into *directories*. Of the six standard pathname components, the host, device and directory components together identify a specific file directory, and the name, type and version components identify one file in the directory.

A *directory listing* is a list of all the files in a directory. ZMACS provides directory listings in brief format (file names only) and verbose format (sizes, dates, and authors included).

M-X List Files

Print a brief directory listing.

C-X C-D Print a verbose directory listing.

To print a brief directory listing, use **M-X List Files**. It accepts a file name pattern (optionally including wild cards) using the minibuffer, then lists all file names that match the pattern. To print a verbose listing, use the command **C-U C-X C-D (Display Directory)**, given a numeric argument with **C-U**). It works the same, except that it displays a full line of information for each file listed.

C-X C-D with no numeric argument prints a verbose directory listing of files related to the one you are visiting. Files are considered related if their names are the same except for the type and version components.

Each entry in a directory listing is mouse-sensitive. A box appears around the entry when the mouse is positioned properly to select it. When this is so, \square causes the identified file to be visited in ZMACS. \square pops up a menu with which you can visit the file, load it (assuming it is a file of LISP code or a compiled QFASL file), or compare its contents against the latest version of the same file (see section 17.7 [Source Compare], page 84).

You can request that ZMACS commands that operate on files automatically display a directory listing describing files related to the one operated on. The variable `zwei:*auto-directory-display*` controls this. Its permissible values are `nil` (the default; no automatic display), `:read` (automatic display after read operations), `:write` (automatic display after write operations), and `t` (automatic display after either read or write operations).

17.6 Deleting and Expunging Files

Once a file is created, it exists forever unless it is *deleted*. You must delete files that are no

longer needed, in order to reclaim their disk space for new files. However, deleting a file that will be needed later can be a disaster. Both file servers and ZMACS provide various features to assist you in deleting useless files without danger of deleting useful ones.

M-X Delete File

Delete one file, or files that match a wildcard pattern.

M-X Undelete File

Undelete one file, or files that match a wildcard pattern.

M-X Expunge Directory

Really erase all deleted files in a directory.

M-X Reap File

Delete old versions of one file.

M-X Clean Directory

Delete old versions of each file in a directory.

The basic ZMACS command for deleting files is **M-X Delete File**. It accepts a file name pattern (optionally including wild cards) using the minibuffer, and deletes all the files that match the pattern. If a single file name is specified, it is deleted and a message is printed in the echo area to report completion:

```
Deleted: OZ:<RMS>FOO.BAR.5
```

If wildcards were specified, **M-X Delete File** prints a list of the actual files that match, and asks for confirmation before proceeding to delete the files.

17.6.1 Expunging and Undeletion

On some file servers, deleting a file does not really erase it, but marks the file as "slated for erasure". Deleted files are actually erased when the containing directory is *expunged*. To do this, use **M-X Expunge Directory**. The directory to expunge must be specified as an argument using the minibuffer; the default is the directory of the currently visited file.

Instead of expunging deleted files, you can *undelete* them using the command **M-X Undelete File**. It is used just like **M-X Delete File**.

Unix, Multics and ITS file servers erase files immediately on deletion. **M-X Expunge Directory** and **M-X Undelete File** give errors when used on files on these types of servers.

17.6.2 Deleting Old Versions

The normal course of editing constantly creates new versions of files (if you are using a file server that supports multiple versions; all except Unix and Multics do). If you don't eventually delete the old versions, you will waste lots of disk space. ZMACS has commands that make it easy to delete the old versions.

M-X Reap File and **M-X Clean Directory** are convenient ways to do the usual thing: keep only the two (or other number) most recent versions.

M-X Reap File reads a file name pattern using the minibuffer, and operates on all files that match the pattern. The default pattern selects all files that are the same as the visited file except for type and version number. For each sequence of file versions, **M-X Reap File** finds which versions are consecutive with the latest version, decides to keep the latest two versions, and offers to delete all the rest. With a single **Y** or **N**, you choose to delete them or retain them. Individual versions whose "don't reap" bits are set (see section 18.4 [Dired Props], page 93) are omitted from the list offered and are never deleted by **M-X Reap File**.

Any file whose type is a member of the list **zwei:*temp-file-type-list*** is also offered for deletion regardless of version number, unless its "don't reap" bit is set.

The variable **zwei:*file-versions-kept*** determines how many of the most recent versions are normally kept. Normally its value is 2. If a numeric argument is given to **M-X Reap File**, it specifies how many versions to keep.

M-X Clean Directory is like **M-X Reap File** except that it applies (by default) to all the files in the same directory as the visited file. The two commands differ only in the default file name patterns; if you specify a file name pattern, you get the same results from either command.

For complete flexibility to delete precisely the files you want to delete, you can use the Dired package. See chapter 18 [Dired], page 91.

17.7 Comparing Files

Source Compare is a ZMACS facility for comparing two files (or ZMACS buffers), saying which lines have to be replaced or changed to transform one file into the other. Here is an example of the output *Source Compare* produces for particular input.

Contents of buffer foo:

```
foo
bar
lose
this
is
a
test
```

Contents of buffer bar:

```
foo
bar
lose
this
was
not
a
test
```

Output from Source Compare:

```
;Source Compare of Buffer foo and Buffer bar on 5/29/85 20:16:37
**** Buffer foo, Line #4
is
a
**** Buffer bar, Line #4
was
not
a
*****
```

The general way to run Source Compare is with **M-X Source Compare**. You must specify two files-or-buffers, by name. First, type **F** if the first file-or-buffer is a file, or **B** if it is a buffer. Then, in the minibuffer, enter the name of the file or buffer. Then type **F** or **B** for the second file-or-buffer, followed by its name. In the above example, comparing buffers named 'foo' and 'bar', you would type

```
M-X Source Compare (RETURN) B foo (RETURN) B bar (RETURN)
```

The results of the comparison are normally printed as typeout. To get them in a buffer, use **M-X Execute Command Into Buffer** first. See section 1.2 [Typeout], page 5.

One special case of Source Compare is handled conveniently by **M-X Source Compare Changes**. No arguments are needed, because this command always compares the current buffer with the file visited in it. It shows you what changes you have made in editing this buffer. If the buffer and the

file match exactly, the buffer's modified-flag is cleared.

Source Compare can also be invoked with the = command in Dired (see chapter 18 [Dired], page 91) or BDired (see section 18.8 [BDired], page 96).

17.7.1 Merging Files

Source Compare can also be used to combine two similar files interactively, resulting in a file that resembles one of the original files here, the other one there, as you wish. This is done with **M-X Source Compare Merge**. It reads arguments specifying which files or buffers to compare, like **M-X Source Compare**, and then reads the name of a buffer to put the merged output into.

Typically this command is used when two people have independently changed one program or document, in order to create a version with both of their changes. Source Compare can only tell where the files differ; it cannot tell which file's version is preferable. The user directing the merge operation must tell it this.

The first stage of merging takes place without interaction. It places merged text in the specified buffer for output. Lines that are common to the two input files are copied only once to the output. Where the input files differ, the output shows both versions, like the output from **M-X Source Compare**. Here is what it would look like for the sample buffers shown above:

```
foo
bar
lose
this
*** MERGE LOSSAGE ***
*** Buffer foo HAS:
is
*** Buffer bar HAS:
was
not
*** END OF MERGE LOSSAGE ***
a
test
```

The second stage of merging is to ask you what should be done for each run of differences between the two inputs. **M-X Source Compare Merge** displays the runs of differences in sequential order through the file, and for each one asks you to type a character to specify an action. This is somewhat like **Query Replace**. Your alternatives are:

- 1 Take the first input file's version of these lines.
- 2 Take the second input file's version of these lines.
- (RUBOUT)** Take neither version of these lines.
- * Take both version of these lines, without the header lines (the lines containing asterisks).
- I Take both version of these lines, and leave the header lines there.

After typing one of these commands, you have a chance to confirm it, retract it, or edit the results. Type **(SPACE)** to confirm and move to the next set of differences. Type **(RUBOUT)** to retract; the text returns to its previous state and you are asked once again what to do with these differences. Type **Control-R** to edit the text with a recursive edit; type **(END)** to exit the recursive edit and continue merging.

These additional options are available as alternatives to 1, 2, I, * and **(RUBOUT)**:

Control-R

Enter recursive edit. You can alter the text shown, then type **(END)** to return to **M-X Source Compare Merge**. Then you must tell it once again what to do with this set of differences. **(SPACE)** is often what you want, then. See section 29.1 [Recursive Edit], page 179.

Control-L

(CLEAR-SCREEN)

These have their usual meanings for redisplaying the screen. They do not tell **M-X Source Compare Merge** what to do with the latest set of differences, so it asks you again.

(SPACE)

Like I, but you do not need to confirm, and have no chance to change your mind.

- ! Process this set of difference and all following differences with no further interaction. You must type a second input character saying how to process them:
 - !1 Take the first input file's version of each difference.
 - !2 Take the second input file's version of each difference.
 - !* Take both version of each difference, without header lines.
 - !I Take both version of each difference, with header lines.

Once each set of differences has been processed with one of these commands, the buffer of output is resectionized, and merging is finished.

17.8 Miscellaneous File Operations

ZMACS has extended commands for performing many other operations on files.

M-X View File allows you to scan or read a file by sequential screenfuls without waiting for the time required to read the entire file into ZMACS. It reads a file name argument using the minibuffer. After opening the file, it reads and displays one window full. You can then type **(SPACE)** to scroll forward one window full, or **(OVERSTRIKE)** (chosen by analogy to the Backspace key on other systems) to scroll back. Those are the only commands available for moving around in the file being viewed; any other character exits **View File** and returns to the usual editing context. **View File** reads more of the file only as necessary to obey scrolling commands.

M-X Print File makes a hardcopy of the specified file, using the LISP function `hardcopy-file` with just the file name as argument. See section 29.3 [Hardcopy], page 180.

M-X Insert File inserts the contents of the specified file into the current buffer at point, leaving point unchanged before the contents and an inactive mark after them. See chapter 10 [Mark], page 39.

M-X Insert File No Fonts is like **M-X Insert File** except that it discards font change information found in a multi-font file. See section 23.2 [Fonts], page 120.

M-X Write Region To File is the inverse of **M-X Insert File**; it copies the contents of the region into the specified file. **M-X Append to File** adds the text of the region to the end of the specified file, and **M-X Prepend to File** adds it to the beginning.

Files of LISP code can also be loaded into the LISP world or compiled. See section 26.2 [Compile File], page 157.

M-X Rename File reads two file names *old* and *new* using the minibuffer, then renames file *old* as *new*. Both names can be file name patterns containing wildcards; in this case, all the files matching *old* are renamed to names constructed according to *new*. See the operation `:translate-wild-pathname` in the *LISP Machine Manual* for the rules used in doing this. If *old* contains wildcards, **M-X Rename File** prints a list of all the renamings to be done and asks for confirmation before doing any.

M-X Copy File works just like **M-X Rename File**, but copies the files *old* to the files *new*. Copying a file requires deciding whether to treat it as a binary file or a text file. **M-X Copy File** guesses this from various information such as the file name type component and whether the file appears to be a QFASL file. It is usually right, but if necessary you can specify this information explicitly by using the command **M-X Copy Text File** or **M-X Copy Binary File** instead. Normally, all copy commands preserve the author's name and creation date; that is, the

new files receive the same author and date as the old files. A numeric argument to any of the copy command prevents this; then the new file receives the current date, and you as the author.

M-X Create Link is a command for creating a symbolic link. It reads the name for the new link and the name to link to. Only some file servers support symbolic links.

M-X Create Directory creates a new file directory with the specified name.

File servers record for each file various *file properties*, including the creation date, last modification date, author, and others. File properties are independent of the contents of the file. **M-X Change File Properties** allows you to change various file properties of the specified file. It presents a choose-variable-values window that shows the values of all alterable file properties. Click on the property value that you wish to change, then type in a replacement value. Click on **Done** to put the changes into effect, or **Abort** to cancel them; either one returns to ZMACS editing.

Most of the commands in this section use the visited file name as the defaults for file name arguments. Those that read two file name arguments use the first file name specified as the defaults for the second one.

However, the commands for copying the region to a file or inserting a file's contents use a different set of defaults, called the *aux defaults*. Each of these commands defaults its argument based on the aux defaults, and then sets the aux defaults to the file that was specified. So the default for each miscellaneous file command is the file used in the previous miscellaneous file command.



18. Dired, the Directory Editor

Dired makes it easy to delete, compile or visit many of the files in a single directory at once. It makes a ZMACS buffer containing a listing of the directory. You can use the normal ZMACS commands to move around in this buffer, and special Dired commands to operate on the files.

There are two ways to invoke Dired. **M-X Dired** is the general way; it accepts a file name pattern in the minibuffer and lists all files that match the pattern. The fast way is **C-X D** (**↑R Dired**). With no argument, it invokes Dired listing all the files in the current default directory. With just **C-U** as argument, it lists all files related to the default file name (matching except for type and version number).

Once invoked, Dired creates a buffer containing a listing of the specified files and selects the buffer for editing. You can switch between this buffer and other ZMACS buffers freely, using **C-X B** and all the other usual commands. Whenever the Dired buffer is selected, certain special commands are provided that operate on files that are listed. The Dired buffer is "read-only", and inserting text in it is not useful, so ordinary printing characters such as **D** and **X** are used as Dired commands. Most of these commands operate on the file described by the line that point is on. Some commands perform operations immediately; others "mark" the file to be operated on later.

Most Dired commands that operate on the current line's file also treat a numeric argument a repeat count, meaning to apply to the files of the next few lines. A negative argument means to operate on the files of the preceding lines, and leave point on the first of those lines.

18.1 Deleting Files with Dired

The primary use of Dired is to mark files for deletion and then delete them.

- D, C-D, K, C-K** Mark this file for deletion, or remove a mark for undeletion.
- U** Remove marks on this line, or mark a deleted file for undeletion.
- (RUBOUT)** Remove marks on previous line, moving point to that line; or mark a deleted file there for undeletion.
- X** Operate on files as requested by their marks.
- H** Mark old file versions for deletions.

You can mark a file for deletion by moving to the line describing the file and typing **D, C-D, K,**

or **C-K**. The deletion mark is visible as a **'D'** at the beginning of the line. Point is moved to the beginning of the next line, so that several **D** commands mark several files for deletion.

The files are marked for deletion rather than deleted immediately because deletion is dangerous on file servers that erase deleted files immediately. Until you direct Dired to delete the marked files, you can remove deletion marks using the commands **U** and **(RUBOUT)**. **U** works just like **D**, but removes marks rather than making marks. **(RUBOUT)** moves upward, removing marks; it is like **U** with numeric argument automatically negated.

If the directory is on a file server that supports undeletion, the directory may contain files already deleted but not yet expunged. Such files are indicated with **'d'**, in lower case, at the beginning of the line. **'d'** is not a mark indicating a requested operation; it is an indication of the file's current status. For deleted files, deletion is not possible, but undeletion is. Therefore, on a deleted file, the **D** and **U** commands play the opposite of their usual roles: **U** marks the file for eventual undeletion, putting in a mark visible as a **'U'** at the beginning of the line, and **D** removes this mark, replacing the **'U'** with the **'d'** status-indicator.

To delete the marked files, type **X**. This command performs all the file operations requested by marking files, but first prints a complete list of what will be done to which files, and asks for confirmation. You may type **Y**, **E**, **N** or **Q**. **Y** means to go ahead and perform the deletions and other operations. **E** is similar but afterward expunges the directories that deleted files are contained in, so that the files are really erased. **E** is offered only if the files are on a file server that does not erase files until they are expunged. **N** returns to editing the Dired buffer without performing any of the operations. **Q** returns to editing some other buffer, the most recent one selected before the Dired buffer was. Use **Q** if you change your mind about using Dired at all.

The **H** command marks many files for deletion, based on their version numbers. **H** looks at the current line's file and all the other listed files differing only in version number and type; it marks all older versions for deletion using the same criteria as **M-X Reap File** (see section 17.6 [Deleting Files], page 82). **C-U H** operates on all the files in the Dired buffer in this way.

18.2 Dired Cursor Motion

All the usual ZMACS cursor motion commands are available in Dired buffers. Some special purpose commands are also provided.

- (SPACE)** Move down one line.
- N** Move down to next file with many versions.

! Move down to next file not backed up on tape.

For extra convenience, **(SPACE)** in Dired is a command similar to **C-N**. Moving down a line is done so often in Dired that it deserves to be easy to type. **(RUBOUT)** is often useful simply for moving up.

The command **N** moves down to the next file that has more than **zwei:*file-versions-kept*** versions (by default, this is 2 versions). The command **!** moves down to the next file whose **'!** flag ("not backed up on tape") is set. The **'!** is visible in the file's line, before the creation date.

18.3 Displaying Other Directories

Most file servers have a hierarchy of directories. For example, if **angel** is a Unix file server, the directory **angel:/lmi/rms** is contained in the directory **angel:/lmi** and might contain the directory **angel:/lmi/rms/bin**. Dired provides commands for moving around in the directory hierarchy.

- <** Invoke Dired on superior directory.
- E** Invoke Dired on subdirectory described on current line.
- S** Include in (or, remove from) this Dired buffer the contents of the subdirectory described on the current line.

The **<** command invokes Dired on the containing directory of the one you are currently looking at. In this example, it would be equivalent to **M-X Dired (RETURN) angel:/lmi/ (RETURN)**. The same effect for a subdirectory can be obtained with the **E** command, if point is on the line describing the subdirectory. These commands create a new Dired buffer for each directory.

The **S** command is used to include the contents of a subdirectory under the line that describes the subdirectory. This makes one Dired buffer display multiple directory levels. The files in the subdirectory can be marked and operated on together with the files in the original directory.

If a subdirectory's files are displayed, the **S** command on the subdirectory's line removes its files from the display. Any request marks on those files are forgotten.

18.4 Operations on File Properties

File servers record for each file various *file properties*, including the creation date, last modifica-

tion date, author, and others. File properties are independent of the contents of the file. Each file can also have *attributes*, which are part of the contents of the file, but assigned a special standard interpretation. See chapter 30 [Attributes], page 187. Dired provides commands for working with both properties and attributes. They all apply to the file described by the line point is on.

- Complement this file's "don't delete" bit. If the file has this bit, any attempt to delete it gets an error.
- Complement this file's "don't supersede" bit. If the file has this bit, any attempt to create a later version of it gets an error.
- Complement this file's "don't reap" bit. If the file has this bit, various automatic old-version deletion programs including **M-X Reap File** and the Dired command **H** will never delete this file. Automatic deletion programs on the file server should also be designed to respect this bit.
- . Change any or all properties of this file.
- . Display the attribute list of this file.

The "don't delete", "don't supersede" and "don't reap" bits are three standard file properties that many, but not all, file servers support. The characters '•', '•' and '•', respectively, appear in the file's line, just before the creation date, to indicate that the file has these properties. The same characters are used as commands to set or clear the bits.

The . command presents a choose-variable-values window with which you can alter any of the alterable properties of the file. Each file server has its own rules for which kinds of properties you can alter.

The , command prints the attribute list of the current line's file. For a text file, this is the information in the attribute line. For a QFASL file, this is a copy of the attribute list of the source file, stored in encoded format. In the case of a QFASL file, , also prints the compilation information, which says who compiled the file, when, where, and from which source version.

There is no command in Dired to alter a file's attribute list. Since the attribute list is part of the file's contents, altering it is done by editing the file.

18.5 Other File Operations in Dired

- A Mark this file for later application of a LISP function.
- F Mark this file to be visited later.

P Mark this file to be printed later.

The **A**, **F** and **P** commands mark files for various operations, much the same way the **D** command marks files for deletion. All the requested operations take place when the **X** command is given.

Marking a file to apply a LISP function may seem obscure, but it is very versatile, because nearly anything can be done to whichever set of files you select. If you have marked any files with **A**, the **X** command reads the name of a LISP function to invoke, using the minibuffer. This function will be called once for each marked file, with the file's pathname object as the sole argument. For example, if the function `compile-file` is specified, each of the marked files is compiled. This is so useful that it is the default, used if you type just `(RETURN)`. You can even type a `lambda`-expression. This way you can cause any LISP program to be run for each **A**-marked file.

Files marked for printing are printed using `hardcopy-file`, with all arguments except the file name defaulted.

The various kinds of Dired marks are exclusive. For example, marking a file for printing removes any mark for deletion, visiting, or function application. Also, the **U** and `(RUBOUT)` commands get rid of any kind of mark except a 'U' (undelete) mark.

18.6 Immediate File Operations in Dired

Some file operations in Dired take place immediately when they are requested.

- C** Copies the file described on the current line. You must supply a file name to copy to, using the minibuffer.
- E** Visits the file described on the current line. It is just like typing **C-X C-F** and supplying that file name. If the file on this line is a subdirectory, **E** actually causes Dired to be invoked on that subdirectory. Note that you can also request files to be visited later using the **F** command.
- C-Shift-E** Like **E**, but creates another window within the ZMACS frame and displays the file's buffer in that window. The Dired buffer remains visible in the first window. See chapter 20 [Windows], page 109.
- L** Loads the file described on the current line, as LISP code, using the function `load`. If the file does not contain LISP source code or compiled LISP code, you will get bizarre results.

- R** Renames the file described on the current line. You must supply a file name to rename to, using the minibuffer.
- V** View the file described on this line using **M-X View File**. Viewing a file is less flexible than editing it, but allows you to see the beginning of the file without waiting for the whole file to be read by ZMACS. See section 17.8 [Other File Operations], page 87.
- Compare the file described on this line against the most recent version of the file, using **M-X Source Compare**. See section 17.7 [Source Compare], page 84.

In addition, most ZMACS commands for operating on files, when used in a Dired buffer, use the file name listed on the current line as the default file name.

18.7 Sorting the Dired Buffer

When you invoke Dired, it creates a buffer sorted in the usual manner for directory listings: increasing alphabetically by file name. Dired provides commands to sort the directory by other file parameters:

- M-X Sort Increasing Creation Date**
Sort by creation date, oldest files first.
- M-X Sort Decreasing Creation Date**
Sort by creation date, newest files first.
- M-X Sort Increasing Reference Date**
Sort by last reference date, less recently used files first.
- M-X Sort Decreasing Reference Date**
Sort by last reference date, most recently used files first.
- M-X Sort Increasing Size**
- M-X Sort Decreasing Size**
Sort files by size.
- M-X Sort Increasing File Name**
Sort in the same order normally used.
- M-X Sort Decreasing File Name**
Sort in the opposite of the usual order.

If files are already marked for operations when sorting is done, the marks move with the files.

18.8 Balancing Directories Interactively

Bdired—Balance DIRectories EDit—provides a way to compare the contents of two file directories, possibly on different machines, and copy files between them.

BDired reads two directory names using the minibuffer, then reads the contents of the directories and matches up corresponding file names. Then it creates a **BDired** buffer, which is like two **Dired** buffers concatenated, one for each directory.

The main thing **BDired** lets you do to a file is transfer it from the directory it is in to the other directory. Files marked for transfer have a 'T' at the beginning of the line. Initially, every file name that is found in one directory and not the other has a 'T' mark. It is easy to remove many of the marks at once using the **U** command with a large numeric argument. Here is a list of **BDired** commands:

- T** Mark the current line's file to be transferred to the other directory (the one it is not already on).
- P** Mark the current line's file to be printed using **hardcopy-file**.
- R** Rename the current line's file to a new name specified using the minibuffer.
- C** Copy the current line's file to a new name specified using the minibuffer.
- U** Cancel any marks for transfer or printing on the current line; if there is none, move up one line and cancel one there.
- (RUBOUT)** Move up one line and cancel any marks for transfer or printing on that line.
- (SPACE)** Move down vertically.

The above commands use a numeric argument as a repeat count. The **U** command with no argument chooses an argument of 1 or -1 based on a heuristic; but, if given an explicit numeric argument, it always goes in the direction specified by the argument.

- Compare the current line's file against the latest version of the file using **Source Compare**. See section 17.7 [**Source Compare**], page 84.
- q**
- (END)** Perform all requested transfers and printouts, then "exit" **BDired** by switching to another buffer.
- (ABORT)** "Exit" by switching to another buffer, without doing any transfers.



19. Using Multiple Buffers

The text you are editing in ZMACS resides in an object called a *buffer*. Each time you visit a file, a buffer is created to hold the file's text. Each time you invoke Dired, a buffer is created to hold the directory listing. Each time you send a message with **C-X M**, a buffer is created to hold the text of the message.

At any time, in one ZMACS frame, one and only one buffer is *selected*. It is also called the *current buffer*. If the ZMACS frame contains multiple windows, then each window can have a chosen buffer; but at any time only one of the windows is selected within the ZMACS frame, and its chosen buffer is the selected buffer. Often we say that a command operates on "the buffer" as if there were only one; but really this means that the command operates on the selected buffer (most commands do).

Each buffer has a name, which can be of any length, and you can select any buffer by giving its name. Most buffers are made by visiting files, and their names are derived from the files' names by permuting the components. A newly started ZMACS frame has a buffer named **Buffer-n** where *n* is an integer numeral. The name of the currently selected buffer is visible in the mode line when you are at top level.

Each buffer records individually what file it is associated with, whether it is modified, and what major mode and minor modes are in effect in it. Any ZMACS variable can be made *local to* a particular buffer, meaning its value in that buffer can be different from the value in other buffers. See section 31.2 [Variables], page 192.

Each ZMACS frame is a separate editor in most respects, but all ZMACS frames share the same set of buffers. Having created a buffer in one ZMACS frame, you can select it for editing in another. This is very important.

19.1 Creating and Selecting Buffers

- C-X B** Select or create a buffer.
- C-M-L** Select another buffer, rotating through several recently selected ones.
- C-X C-M-L**
 Like **C-M-L** but remembers the last numeric argument explicitly given it.
- C-X C-B** List the existing buffers.

To select the buffer named *bufname*, type **C-X B *bufname* RETURN**. This is the command **Select Buffer** with argument *bufname*. Because completion is provided for buffer names, you can abbreviate the buffer name. Most often the buffer name begins with the name component of a file name. Unless you have visited two files with the same name component, the name or an abbreviation for it is enough to identify a buffer for **C-X B** or any other command that reads a buffer name. See section 7.1 [Completion], page 30. An empty argument to **C-X B** specifies the previously selected buffer.

Each ZMACS window records a *buffer selection history*, which contains a list of all the buffers known to ZMACS, ordered with the buffers most recently selected *in that window* coming earlier on the list. The command **C-M-L (Select Previous Buffer)** uses this history to cycle through two or more recently selected buffers.

The simplest case is **C-M-L** without a numeric argument: this selects the next-to-last selected buffer. Repeated use with no other intervening buffer-selecting commands alternates between two buffers.

When **C-M-L** is given a positive numeric argument *n*, it selects the *n*th most recently selected buffer. If this command is repeated, it cycles through all of the last *n* buffers selected. A negative argument *-n* cycles through the same set of buffers in the opposite order. This selects the next-to-last selected buffer regardless of the value of *n*, but it alters the record of recently selected buffers in a peculiar way: it puts the buffer being deselected into the *n*'th place.

C-U 1 C-M-L selects the next-to-last selected buffer but puts the buffer being deselected at the end of the list. Repeating this command cycles through all the buffers known to ZMACS. **C-U -1 C-M-L** also cycles through all known buffers, but in the opposite order.

The command **C-X C-M-L (Select Default Previous Buffer)** does the same thing as **C-M-L** when an explicit numeric argument is given. Without an argument, it differs by using the last argument explicitly given it. **C-X C-M-L** with just a minus sign as an argument uses minus the last argument explicitly given it. **C-M-L**, by contrast, always uses 2 as an argument if none is given.

To print a list of all the buffers that exist, type **C-X C-B (List Buffers)**. Each buffer's name, major mode, and file version number are printed. '*' at the beginning of a line indicates the buffer is "modified". If several buffers are modified, it may be time to save some with **M-X Save All Files** (see section 17.3 [Saving], page 79). '+' indicates a buffer that is associated with a new file, one that does not exist but will be created if you save. '≡' indicates a read-only buffer; these are usually buffers made by **Dired** or similar commands. Here is an example of a buffer list:

Buffers in ZWEI:

Buffer name:	File Version:	Major mode:
♦ foo.bar /lmi/rms/ ANGEL:		(LISP)
* lose	[1 Line]	(LISP)
ZMACS.LISP#> L.ZWEI; DJ:	(534)	(LISP)
Buffer-1	[1 Line]	(LISP)
debug trace	[1 Line]	(LISP)
* *Definitions*	[7 Lines]	(Possibilities)

♦ means new file. * means buffer modified.

Note that the Unix file has no version number, and the non-file buffers give the number of lines instead of a version number.

Each line in the buffer list is mouse-sensitive. When the mouse is properly pointing at a line, a box appears around it; then **[L]** selects the buffer described by that line, and **[R]** pops up a menu containing several useful buffer operations, mostly commands described in this chapter.

Most buffers are created by visiting files, but you can also create a buffer explicitly by typing **C-X B bufname RETURN RETURN**. This is the same as above, with an extra **RETURN**, which is needed because the minibuffer refuses to accept a nonexistent buffer name on the first **RETURN**. This makes a new, empty buffer that is not associated with any file, and selects it for editing. Such buffers are used for making notes to yourself. If you try to save one, you are asked for the file name to use. The new buffer's major mode is determined by the value of **zwei:*default-major-mode*** (see chapter 21 [Major Modes], page 113).

Note that **C-X C-F**, and any other command for visiting a file, can also be used to switch buffers. See chapter 17 [Files], page 75.

19.2 Specifying a Buffer with a File Name

Whenever ZMACS uses the minibuffer to read a ZMACS buffer name, you have the option of specifying a file name instead. ZMACS will visit the file if that has not already been done, and then use the file's buffer.

To do this, you must first type **C-Shift-F**. This character, in the minibuffer, is specially defined to read a file name, visit the file, and then use the file's buffer.

19.3 Miscellaneous Buffer Operations

C-X C-Q Toggle read-only status of buffer.

M-X Rename Buffer

Change the name of the current buffer.

A buffer can be *read-only*, which means that commands to change its text are not allowed. Normally, only buffers made by subsystems such as Dired are read-only. If you wish to make changes in a read-only buffer, use the command **C-X C-Q (Toggle Read Only)**. It makes a read-only buffer writable, and makes a writable buffer read-only.

M-X Rename Buffer changes the name of the current buffer. Specify the new name as a minibuffer argument. There is no default.

The commands **C-X A (Append To Buffer)** and **M-X Insert Buffer** can be used to copy text from one buffer to another. See section 11.3 [Copying], page 50.

19.4 Killing Buffers

After you use ZMACS for a while, you may accumulate a large number of buffers. You may then find it convenient to eliminate the ones you no longer need. There are several commands provided for doing this.

C-X K Kill a buffer.

M-X Kill Some Buffers

Offer to kill each buffer, one by one.

C-X K (Kill Buffer) kills one buffer, whose name you specify in the minibuffer. The default, used if you type just **(RETURN)** in the minibuffer, is to kill the current buffer. If the current buffer is killed, you must specify the name of another buffer to select instead. The last buffer on the buffer selection history serves as a default for this. The killing does not actually take place until you have typed the new buffer name, so this serves as a kind of confirmation. If the buffer being killed is modified (has unsaved editing) then you are asked whether to save it first.

The command **M-X Kill Some Buffers** asks about each buffer, one by one. An answer of **Y** means to kill the buffer. Killing the current buffer or a buffer containing unsaved changes asks for additional information or confirmation like **Kill Buffer**.

19.5 Operating on Several Buffers

There are several ZMACS commands that present a list of all ZMACS buffers and allow you to specify various operations to be performed on them.

M-X Kill Or Save Buffers

Present menu of buffers to save, kill, compile or clear modification flag of.

M-X Buffer Edit

Present editor buffer describing existing buffers, in which you can specify operations to be performed on them.

The easiest way to get a menu of buffers to operate on is to print a list of them with **C-X C-B**, and then request an operation by clicking the mouse on a buffer name. This provides many alternatives for operating on a buffer, but it usually does not allow you to do more than one thing, because the first thing you do will probably redisplay the screen and make the buffer list go away. This section describes other ZMACS commands are provided that allow you to queue up requests for several operations on various buffers.

If you like to use a window-oriented interface and choose operations with the mouse, use the command **M-X Kill Or Save Buffers**. It displays a multiple choice window with a line for each ZMACS buffer (though you may need to scroll it to see all the buffers, if there are many), containing four choice-boxes for saving the buffer, killing the buffer, marking the buffer unmodified, and recompiling the buffer's associated file (offered for LISP source files only). Saving is initially selected for all modified file-visiting buffers.

Click on a choice box to select that operation, or to cancel a selection already made. Un-modifying is not allowed together with saving or killing, but aside from that any combination of operations is allowed. Finally, click on the **Done** box to perform all selected operations, or click on the **Abort** box to return to editing and perform no operations. You can also return to editing by typing **C-(ABORT)**.

If you like to use ZMACS editing to specify operations to be performed, use **M-X Buffer Edit**, a sort of "Dired for buffers". This command creates and selects a ZMACS buffer containing one line for each previously existing ZMACS buffer. You then move around in the buffer using normal ZMACS cursor motion commands, and request operations on buffers by typing commands that apply to the current line. There are four kinds of operations you can request for the buffer described on the current line:

1. Kill the buffer. Type **K** to request this. The request shows as a 'K' on the line, before

the buffer name. If the buffer is modified, this also makes a save request, which shows up as an 'S'.

2. Save, revert or write the buffer, or clear its "modified" flag. Only one of these operations is allowed. They are requested by the commands **S**, **R**, **W** and **~**; the same characters appear in the line as marks to indicate that the operations have been requested. The **W** operation requires a file name argument that specifies the file to write into (using **Write File**); this file name appears in the line after the buffer name.
3. Print the buffer. Type **P** to request this. The request shows as a 'P' on the line, before the buffer name.
4. Select the buffer. The command to request this is **.** (period), and the request is indicated by a period before the buffer name. As only one buffer can be selected, the **.** command clears the period marks from all other lines.

All the operation-requesting commands move down a line.

The command **U** cancels any request (except selection) for the current line, and moves down; **RUBOUT** does so for the previous line, and moves up to it. The command **N** cancels any I/O request (save, revert, write or unmodify) but not kill, print or select requests.

To perform the operations requested, type **Q**. This ends by selecting the buffer marked with the period.

All that **M-X Buffer Edit** does directly is create and select a suitable buffer. Everything else described above is implemented by specially redefined editing commands provided for that buffer. One consequence of this is that you can switch from the **Buffer Edit** buffer to another ZMACS buffer, and edit it. You can reselect the **Buffer Edit** buffer later, to perform the operations already requested, or you can kill it, or pay no further attention to it.

19.6 Buffer Groups

A *buffer group* or *tag table* is a set of buffers, with an ordering. Grouping several related files, such as the source files of one program, makes it possible to search or replace through all the files with one command.

19.6.1 Creating and Selecting Buffer Groups

Several buffer groups can exist, but only one is selected; all the commands to search the buffers in a group use the selected group. The commands to create a buffer group also select it. Buffer groups have names and you can reselect an existing one by specifying its name.

M-X Select All Buffers As Tag Table

Make a buffer group of all existing ZMACS file-visiting buffers, and select it.

M-X Select Some Buffers As Tag Table

Make a buffer group of some existing ZMACS file-visiting buffers, querying buffer by buffer, and select it.

M-X Select System As Tag Table

Make a buffer group of all source files of a system, and select it.

M-X Visit Tag Table

Read a tag table file, make a buffer group of all source files it mentions, and select it.

M-X List Tag Tables

List the names of all the buffer groups.

M-X Select Tag Table

Select a buffer group by name.

M-X Select All Buffers As Tag Table creates and selects a buffer group containing all the existing ZMACS buffers that are associated with files. It receives a generated unique name. A numeric argument tells this command to read a string and then consider only buffers whose names contain the string.

M-X Select Some Buffers As Tag Table is similar, but queries about each buffer to find out whether to include it in the buffer group. In addition to Y or N, you can answer F, which means to make the buffer group using the buffers you have already said yes to, and not ask you about any others.

M-X Select System As Tag Table creates and selects a buffer group containing all the source files in a specified *system* (see the chapter "Maintaining Large Systems", in the *LISP Machine Manual*, for information on what a system is). Since perhaps not all of those files have been visited in ZMACS yet, some of the buffers "in" this group may not exist yet. They are *virtual* members of the buffer group. If you proceed to search or replace through all the buffers in the group, these virtual members will be converted to real members (by visiting the appropriate files) as they are needed.

M-X Visit Tag Table reads in a tag table file and creates and selects a buffer group containing all the source files mentioned in the tag table file. For source files not yet visited, virtual members are made. The tag table file's name is used as the buffer group's name. a Tag table file describes

all the definitions in several program files, and is made with the tools **TAGS** (on Twenex) or **etags** (on Unix).

M-X List Tag Tables prints a list of all defined buffer groups, giving the name of each group and the buffers/files that belong to it. **M-X Select Tag Table** reads the name of a buffer group and selects that one.

19.6.2 Searching a Group of Buffers

M-X Tags Search

Search for the specified string within buffers in the selected group.

M-X Tags Query Replace

Perform a **Query Replace** within buffers in the selected group.

M-X Tags Multiple Query Replace

Perform a **Query Replace** within buffers in the selected group.

M-X Tags Multiple Query Replace From Buffer

Perform a multiple **Query Replace** from the contents of the specified buffer

C-. Restart one of the commands above, from the current location of point.

M-X Tags Search searches the buffers of the selected group, one by one, for a specified string. Extended search characters can be used in the string (See section 15.7 [Extended Search], page 68). Search starts at the beginning of the first buffer in the group and continues with additional buffers until a match is found. At that time, the buffer containing the match is selected and point is put after the match. The previous location of point is pushed on the point pdl.

Virtual members of the group (files not yet visited) are visited when it is time to search them.

Having found one match, you probably want to find all the rest. Type **C-.** to resume the **M-X Tags Search**, searching the rest of the current buffer, followed by the remaining buffers of the group.

M-X Tags Query Replace performs a single **Query Replace** through all the buffers of the group, in their proper order. It reads a string to search for and a string to replace with, just like ordinary **M-X Query Replace**. It searches much like **M-X Tags Search** (sorry, no extended search characters allowed) but repeatedly, processing matches according to your input. See section 15.5 [Replace], page 65, for more information on **Query Replace**.

It is possible to get through all the buffers in the group with a single invocation of **M-X Tags Query Replace**. But since any unrecognized character causes the command to exit, you may need to continue where you left off. **C-** can be used for this. **C-** resumes the last buffer group search or replace command that you did.

M-X Tags Multiple Query Replace and **M-X Tags Multiple Query Replace From Buffer** are buffer group variants of the single-buffer-searching commands **M-X Multiple Query Replace** and **M-X Multiple Query Replace From Buffer**. See section 15.5 [Replace], page 65.

19.6.3 Stepping Through a Buffer Group

If you wish to process all the buffers in a group, but **M-X Tags Search** and **M-X Tags Query Replace** in particular are not what you want, you can use **M-X Next File**.

M-X Next File

Select the next buffer in the selected buffer group.

C-U M-X Next File

With a numeric argument, regardless of its value, select the first buffer in the group.

19.6.4 Buffer Groups and Sectionization

Various commands operate on particular sections in all the buffers in the selected buffer group. See section 26.1 [Sectionization], page 153.

M-X Tags Search List Sections

List sections containing specified string in buffers in the selected group.

M-X Tags Edit Changed Sections

Edit any modified sections in buffers in the selected group.

M-X Tags List Changed Sections

List any modified sections in buffers in the selected group.

M-X Tags Evaluate Changed Sections

Evaluate any modified sections in buffers in the selected group.

M-X Tags Compile Changed Sections

Compile any modified sections in buffers in the selected group.

M-X Tags Add Patch Changed Sections

Add to current patch file any modified sections in buffers in the selected group.

M-X Tags Search List Sections searches the buffers in the group like **Tags Search**; for each match, it records the section that the match occurred in. Finally, the names of the sections found are printed, and a possibilities list is made containing them all. You can use this to visit the sections one by one, later. See section 26.7 [Possibilities Lists], page 164.

The other commands in this section operate on the *changed* sections in the buffers of the group. These commands have variants that operate on the changed sections of one buffer, or of all buffers. The **Tags** variant is midway in scope between those two. For example, **M-X Tags List Changed Sections** operates on more buffers than does **M-X List Buffer Changed Sections** (which scans only the current buffer), but fewer than does **M-X List Changed Sections** (which scans all buffers).

M-X Tags List Changed Sections and **M-X Tags Edit Changed Sections** set up a possibilities list containing all the changed sections. See section 26.7 [Possibilities Lists], page 164.

20. Multiple Windows

You can split a ZMACS frame into two ZMACS windows, which can display parts of different buffers, or different parts of one buffer. You may have more than two windows, but most of the commands are designed for splitting into two windows.

- C-X 2** Start showing two windows. If already doing so, just select the other window.
- C-X 3** Show a second window but leave the first one selected.
- C-X 0** Switch to the Other window (O, not zero).
- C-X 1** Show only one window again.
- C-X 8** Show two windows, both displaying the current buffer, with the top window just big enough to show the current region.
- C-X 4** Find a buffer, file or LISP function in the other window.
- C-X -** Make the selected window bigger, at the expense of the other(s).
- C-M-V** Scroll the other window.
- M-X Split Screen**
Show several windows, with contents specified using a menu.

When multiple windows are being displayed, each window has a ZMACS buffer designated for display in it. The same buffer may appear in more than one window; if it does, any changes in its text are displayed in all the windows where it appears. But the windows showing the same buffer can show different parts of it, because each window has its own value of point.

At any time, one of the ZMACS windows within the frame is the *selected window* within the frame; the buffer this window is displaying is the current buffer. The cursor in this window blinks (provided the ZMACS frame is selected); the cursors in the other ZMACS windows are nonblinking. Commands to move point affect the value of point for the selected ZMACS window only. They do not change the value of point in any other ZMACS window, even one showing the same buffer. The same is true for commands such as **C-X B** to change the selected buffer; they affect only the selected window.

The mode line describes the status of the selected window only. When there are multiple windows, each window has a *label*, a line of text just above the bottom edge, which states the name of the buffer displayed in the window. When there is only one window, it has no label, because in this case it would give no more information than the mode line does.

Once you have used two windows in a ZMACS frame, the frame always remembers both of them. If you change back to using the entire frame for a single window, the other window is kept

dormant. If you resume displaying two windows again, the other window comes back with its former contents.

The command **C-X 2 (Two Windows)** begins displaying two windows within the ZMACS frame. Assuming that you give this command at a time when you have only one window, that window shrinks to the top half of the frame, while the other window appears below and becomes selected. If this is the first time you have used two windows in this ZMACS frame, the other window starts out displaying the same buffer as the first one, with the same value of point. But once the two windows exist, switching buffers or moving point in one of them does not affect the other.

To select the other window, use **C-X 0 (Other Window)**. That is an O, not a zero. **C-X 0** is actually a more general command: when you are using more than two windows, it selects the window beneath the one now selected; or the one at the top, if the lowest one is now selected. If given a numeric argument, it moves the selection that many windows down.

The **C-X 3 (View Two Windows)** command is like **C-X 2** except that it does not change the selected window. That is, it makes a second window appear, but the old one remains selected. **C-X 2** is equivalent to **C-X 3 C-X 0**. **C-X 3** is equivalent to **C-X 2 C-X 0**, but **C-X 3** is much faster.

The selected window can be scrolled using the usual scrolling commands (see chapter 14 [Display], page 59), and each window can be scrolled using its own scroll bar. In addition, the command **C-M-V (Scroll Other Window)** scrolls the window beneath the selected one in cyclic order (so in the bottommost window it operates on the topmost one). It works just like **C-V**, but operates on a different window. **C-M-V** is very useful when you are editing in one window while using the other just for reference.

To return to viewing only one window, use the command **C-X 1 (One Window)**. One of the windows expands to fill the whole ZMACS frame, and the others become dormant. There are various options to control which window remains visible. If **C-X 1** is given an argument, as in **C-U C-X 1**, then it is always the selected window. Otherwise, what happens depends on the variable `zwei:one-window-default`. It has these values:

:current Keep the selected window. This is the default.
:other Keep the uppermost nonselected window.
:upper Keep the uppermost window.
:lower Keep the lowermost window.

In any case, a window displaying compiler warnings will not be chosen for retention by **C-X 1** with no argument.

C-X 0 is meaningful when viewing only one window, if you have used two windows at some time in the past. It displays the other, dormant window, making the previously displayed window dormant. If there is one visible window before **C-X 0**, there is again one visible window afterward, but it is the *other* window.

C-X 2 and **C-X 3** divide the ZMACS frame equally among the two windows. You can redistribute screen space between the windows with the **C-X ^ (Grow Window)** command. It makes the currently selected window get one line bigger, or as many lines as is specified with a numeric argument. With a negative argument, it makes the selected window smaller. The division of space between the windows is remembered after **C-X 1** and comes back when the next **C-X 2** is done.

Another way to divide the frame unevenly into two windows is **C-X 8 (Two Windows Showing Region)**. A region must exist when this command is used. It displays the region in the upper window, making that window exactly big enough to show the whole region (unless the region is too big for that); and displays the same buffer in the lower window, selecting that window. You can then edit elsewhere in the buffer using the lower window, while the specified region remains visible above for your reference.

A convenient "combination" command for viewing something in the other window is **C-X 4 (Modified Two Windows)**. With this command you can ask to see any specified buffer, file or tag in the other window. Follow the **C-X 4** with either **B** and a buffer name, **F** or **C-F** and a file name, or **T** or **.** and a section name (the name of a LISP function, variable or whatever; see section 26.1 [Sectionization], page 153) This selects the other window and displays there the buffer that you specified, such as the buffer containing the specified file. If there is already another window, it is used; otherwise, a second window is made and used.



21. Major Modes

ZMACS has many different *major modes*, each of which customizes ZMACS for editing text of a particular sort. The major modes are mutually exclusive, and each buffer has one major mode at any time. The mode line normally contains the name of the current major mode, in parentheses. See section 1.4 [Mode Line], page 7.

The least specialized major mode is called *Fundamental mode*. This mode has no mode-specific redefinitions or variable settings, so that each ZMACS command behaves in its most general manner, and each option is in its default state. For editing any specific type of text, such as LISP code or English text, you should switch to the appropriate major mode, such as LISP mode or Text mode.

Selecting a major mode changes the meanings of a few commands to become more specifically adapted to the language being edited. Most commands remain unchanged; the ones that usually change are `(TAB)`, `(RUBOUT)`, and `(LINEFEED)`. In addition, the commands that handle comments use the mode to determine how comments are to be delimited. Many major modes redefine the syntactical properties of characters appearing in the buffer. See section 31.5 [Syntax], page 201.

The major modes fall into three major groups. LISP mode, Macsyma mode, TECO mode, MIDAS mode, PL1 mode and C mode are for specific programming languages. Text mode is for editing straight English text, and Tex mode and Bolio mode are for text intended to be passed to text formatters T_EX and Bolio. The remaining major modes are not intended for use on user's files; they are used in buffers created for specific purposes by ZMACS, such as Dired mode for buffers made by Dired (see chapter 18 [Dired], page 91), and Mail mode for buffers made by **M-X Mail**.

Selecting a new major mode is done with an **M-X** command. From the name of a major mode, add '**Mode**' to get the name of a command function to select that mode. Thus, you can enter LISP mode by executing **M-X LISP Mode**.

When you visit a file, usually the file name's type component tells ZMACS to select a major mode automatically. For a file of generic type `:lisp`, such as `TWX:<RMS>FOO.LISP` or `unx:/u/rms/foo.l`, ZMACS automatically puts the buffer in LISP mode. The correspondence between file types and major modes is set by the variable `fs:*file-type-mode-alist*`, whose elements look like `(type . mode-keyword)`. `type` is a type component, either a string or a generic type keyword such as `:lisp`, `:text` or `:doc`. `mode-keyword` is a keyword that identifies a major mode—`:lisp`, `:text`, `:fundamental`, and so on. Often `type` and `mode-keyword` are the same.

You can specify which major mode should be used for editing a certain file by putting a **Mode**

attribute in the file's attribute list. For example,

```
;-*-Mode: Lisp;-*-
```

tells ZMACS to use LISP mode. An explicit **Mode** attribute overrides any assumptions based on the file's type.

If a file's **Mode** attribute is not the name of a ZMACS major mode, the file can still be visited, but the default major mode is used and an error is reported. You can translate such **Mode** attributes into major modes that ZMACS does have, using the variable `zwei:*major-mode-translations*`. Each element of this variable looks like (*spec-mode-keyword . use-mode-keyword*). Thus, (`:scribe . :text`) as an element of this list says to use Text mode for a file whose **Mode** attribute is 'Scribe'.

When a file is visited that does not specify a major mode to use, or when a new buffer is created with **C-X B**, the major mode used is that specified by the variable `zwei:*default-major-mode*`, which is normally `:lisp` for LISP mode. If `zwei:*default-major-mode*` is `nil`, the major mode is taken from the previous selected buffer.

Most programming language major modes specify that only blank lines separate paragraphs. This is so that the paragraph commands remain useful. See section 24.4 [Paragraphs], page 128. They also cause Auto Fill mode to use the definition of **(TAB)** to indent the new lines it creates. This is because most lines in a program are usually indented. See chapter 22 [Indentation], page 115.

22. Indentation

- (TAB)** Indent "appropriately" in a mode-dependent fashion.
- M-(TAB)** Insert a tab character.
- (LINEFEED)**
- Perform **(RETURN)** followed by **(TAB)**.
- M-^** Merge two lines. This would cancel out the effect of **(LINEFEED)**.
- C-M-O** Split line at point; text on the line after point becomes a new line indented to the same column that it now starts in.
- M-O** Indent a new line after this one, or indent this line, to column determined by point.
- M-M** Move (forward or back) to the first nonblank character on the current line.
- M-I** Indent to tab stop. In Text mode, **(TAB)** does this also.
- C-M-** Indent several lines to same column.
- C-X (TAB)** Shift block of lines rigidly right or left.

Most programming languages have some indentation convention. For LISP code, lines are indented according to their nesting in parentheses. The same general idea is used for C code, though many details are different. For assembler code, almost all lines start with a single tab, but some have one or more spaces as well.

Whatever the language, to indent a line, use the **(TAB)** command. Each major mode defines this command to perform the sort of indentation appropriate for the particular language. In LISP mode, **(TAB)** aligns the line according to its depth in parentheses. No matter where in the line you are when you type **(TAB)**, it aligns the line as a whole. In MIDAS mode, **(TAB)** inserts a tab, that being the standard indentation for assembly code. PL1 mode knows in great detail about the keywords of the language so as to indent lines according to the nesting structure.

In Text mode, **(TAB)** runs **Tab To Tab Stop**, which indents to the next tab stop column. This command is available as **M-I** in all modes. You can set the tab stops with **M-X Edit Tab Stops**.

If you just want to insert a tab character in the buffer, you can use **M-(TAB) (Insert Tab)** or **C-Q (TAB)**.

ZMACS normally uses both tabs and spaces to indent lines. If you prefer, all indentation can be made from spaces only. To request this, set **zwei:*indent-with-tabs*** to nil.

For English text, usually only the first line of a paragraph should be indented. So, in Text mode,

new lines created by Auto Fill mode are not indented. LISP mode and other program editing modes tell Auto Fill mode to indent new lines by setting the variable `zwei:space-indent-flag*` to `t`.

To move over the indentation on a line, do **Meta-M** or **C-M-M (Back To Indentation)**. These commands, given anywhere on a line, position the cursor at the first nonblank character on the line.

To insert an indented line before the current line, do **C-A C-O (TAB)**. To make an indented line after the current line, use **C-E (LINEFEED)**.

C-M-O (Split Line) moves the text from point to the end of the line vertically down, so that the current line becomes two lines. **C-M-O** first moves point forward over any spaces and tabs. Then it inserts after point a newline and enough indentation to reach the same column point is on. Point remains before the inserted newline; in this regard, **C-M-O** resembles **C-O**.

M-O (This Indentation) is a command for making a new indented line after the current line or altering the indentation of the current line. The amount of indentation it inserts is enough to reach the column point was in before the **M-O** command.

M-O Inserts a new line after the current line, empty aside from enough indentation to reach the column that point was in before the **M-O**. Point moves to the end of the new line.

M-O M-O **M-O** with a numeric argument adjusts the indentation of the current line so that it reaches to the column point was in before the **M-O**. Point moves to the end of that indentation; thus, it remains in the same spot on the screen (or close to it, if variable width fonts are in use) as the text changes.

To join two lines cleanly, use the **Meta-~ (Delete Indentation)** command to delete the indentation at the front of the current line, and the line boundary as well. They are replaced by a single space, or by no space if at the beginning of a line or before a ')' or after a '('. To delete just the indentation of a line, go to the beginning of the line and use **Meta-\ (Delete Horizontal Space)**, which deletes all spaces and tabs around the cursor.

There are also commands for changing the indentation of several lines at once. **Control-Meta-\ (Indent Region)** gives each line that begins in the region the "usual" indentation by invoking **(TAB)** at the beginning of the line. A numeric argument specifies the indentation, and each line is shifted left or right so that it has exactly that much. **C-X (TAB) (Indent Rigidly)** moves all of the lines in the region right by its argument (left, for negative arguments). The whole group of lines move rigidly sideways, which is how the command gets its name.

22.1 Tab Stops

For typing in tables, you can use Text mode's definition of **(TAB)**, **Tab To Tab Stop**. This command inserts indentation before point, enough to reach the next tab stop column. If you are not in Text mode, this function can be found on **M-I** anyway.

Set the tab stops using **M-X Edit Tab Stops**, which allows you to edit some text that defines the tab stops. When you are finished changing the text, type **(END)**, which exits from **M-X Edit Tab Stops** and makes the changed tab stops take effect. It also returns you to your previous editing.

Here is what the text representing the tab stops looks like for ordinary tab stops every eight columns.

```

:      :      :      :      :      :

```

The second line contains a colon or period at each tab stop. Colon indicates an ordinary tab, which fills with whitespace; a period specifies that characters be copied from the corresponding columns of the first line above it. Thus, you can tab to a column automatically inserting dashes or periods, etc. It is your responsibility to put in the first line the text to be copied. In the example above there are no periods, so the first line is not used, and is left blank.

Normally, a tab character in the buffer is displayed as whitespace that extends to the next display tab stop position, and display tab stops come at intervals equal to eight spaces in the first (or only) font. (How the character tab in the buffer is displayed has nothing to do with the definition of the **(TAB)** character as a command.) The number of spaces per tab is controlled by a file attribute, the **Tab Width** attribute (see chapter 30 [Attributes], page 187). For example, if a file's attribute line contains **'Tab Width: 10;'** then display tab stops will come ten space-widths apart for that file. This is useful for displaying files brought from other operating systems whose normal tab stop spacing is not eight. The command **M-X Set Tab Width** can be used to change the display tab width (in space-widths) of any buffer at any time.

22.2 Other Styles of Indenting a Line

There are several ZMACS commands that indent the current line in particular ways. Some of these are connected to character commands in certain major modes.

M-X Indent Under

Indent to match last occurrence of specified string.

M-X Indent Relative

Indent to an indentation point in the previous line.

M-X Indent Nested

Indent to a previously used indentation level.

M-X Indent Under reads a string using the minibuffer and then indents to match a previous by occurrence of that string in the buffer. It searches back line by line for a line for an occurrence of the string that is farther from left margin than point currently is. The beginning of that match is the place to indent under. It deletes all whitespace from around point and then indents to that column.

M-X Indent Relative indents at point based on the previous line (actually, the previous nonempty line.) It inserts whitespace at point, moving point, until it is underneath an indentation point (the end of a sequence of whitespace) in the previous line. The end of the previous line counts as one.

If point is already indented farther than all the indentation points in the previous line, the whitespace before point is deleted and then the first indentation point then applicable is used.

If no indentation point in the previous line is applicable in any case, then **M-X Tab To Tab Stop** is used instead. The same thing is done if a numeric argument is given. This is so that if a major mode defines **(TAB)** as **M-X Indent Relative**, it is still easy to use **M-X Tab To Tab Stop**.

M-X Indent Nested is intended for editing code in languages for which ZMACS cannot tell where levels of nesting begin and end. It alters the indentation on the current line, regardless of where point is on the line, and point remains fixed in the text of the line. Normally, it indents to match the previous line that is not a comment.

A numeric argument greater than 1 means that the line should be indented at a previous nesting level. Nesting levels are defined by scanning backwards, line by line; each time a line is found that is indented less than all the following lines, that is a different nesting level. (Lines containing just comments and whitespace are ignored.) Thus, **M-2 M-X Indent Nested** would indent to the nesting level just outside the one previously in use.

23. Case and Fonts

In ZMACS, each character can be in any of the available fonts, just as each letter can be upper case or lower case. In programs just as in English text, there are conventions for the use of case and fonts to emphasize the information conveyed by the characters themselves. The ZMACS commands for manipulating case and fonts are designed to work with these conventions.

23.1 Case Conversion Commands

ZMACS has commands for converting either a single word or any arbitrary range of text to upper case or to lower case. There are also case conversion commands specifically for LISP code (see section 25.7 [Lisp Case], page 146).

- M-L** Convert following word to lower case.
- M-U** Convert following word to upper case.
- M-C** Capitalize the following word.
- C-X C-L** Convert region to lower case.
- C-X C-U** Convert region to upper case.

The word conversion commands are the most useful. **Meta-L (Lowercase Word)** converts the word after point to lower case, moving past it. Thus, successive **Meta-L** commands convert successive words. **Meta-U (Uppercase Word)** converts to all capitals instead, while **Meta-C (Uppercase Initial)** puts the first letter of the word into upper case and the rest into lower case. All these commands convert several words at once if given an argument. They are especially convenient for converting a large amount of text from all upper case to mixed case, because you can move through the text using **M-L**, **M-U** or **M-C** on each word as appropriate, occasionally using **M-F** instead to skip a word.

When given a negative argument, the word case conversion commands apply to the appropriate number of words before point, but do not move point. This is convenient when you have just typed a word in the wrong case. You can give the case conversion command and continue typing.

If a word case conversion command is given in the middle of a word, it applies only to the part of the word that follows point. This is just like what **Meta-D** does. With a negative argument, it applies only to the part of the word before point.

The other basic case conversion commands are **C-X C-U (Uppercase Region)** and **C-X C-L (Lowercase Region)**, which convert everything between point and mark to the specified case. Point and

mark do not move.

23.2 Fonts

ZMACS supports multiple fonts within one file, both for English text and for LISP programs. Each ZMACS buffer records a list of fonts that can be used in it. When you visit a file, the file's buffer's list of fonts is determined from the file's **Fonts** attribute (see chapter 30 [Attributes], page 187). You can also specify it explicitly with the **M-X Set Fonts** command.

Each character in the buffer contains a font number, which identifies a font in the list by its position in the list. There are several commands to change the font of characters already in the buffer. You can also specify which font should be used for newly inserted text.

23.2.1 Specifying the List of Fonts

ZMACS records a list of font names for each buffer. In the simplest case, the list is empty. This means that the buffer does not specify fonts; the characters in the text do not contain font numbers, and ZMACS displays all of them using the current default font for window display. If the font list is not empty, then the buffer is a *multi-font* buffer, and each character in it contains a number specifying a font from the list. If a character specifies "the third font", then it is always displayed using the font that is the third in the list.

Usually, you specify fonts for a file with a **Fonts** attribute, as in

```
:: --Mode: Text; Fonts: TR10, HL10, TR10I;--
```

which specifies three fonts. When ZMACS visits the file, it sets the file's buffer's font list according to the file's **Fonts** attribute. If a file has no **Fonts** attribute, ZMACS makes a buffer for it that does not specify a font. Buffers created with **C-X B**, and those made for special purposes such as **Dired**, also normally do not specify fonts.

You can also set a buffer's font list explicitly with the command **M-X Set Fonts**. Using the minibuffer, enter zero or more font names separated by commas (and optionally spaces as well). Specifying no font names makes a non-multi-font buffer. This command asks you whether to modify the **Fonts** attribute in the attribute line to match the list you have just entered. Doing this is a way of changing the file's font list "permanently" (assuming you save the file with the changes

in its attribute list). If you decline to change the attribute line, the new font list is necessarily “temporary”; it applies to the current editing session only.

In addition to the font list, you may wish to specify the **Vsp** attribute, which says how many blank pixels to leave between lines. By default it is 2, which looks good with the default font, but larger fonts may require larger **Vsp** values. The **Vsp** attribute can be set with ‘**Vsp: number;**’ in the attribute line or with the **M-X Set Vsp** command. See chapter 30 [Attributes], page 187.

You may have wondered what happens if you use **Set Fonts** to specify a list of four fonts in a buffer that contains characters that claim to be in the fifth font. The answer is that these characters continue to remember their font number, but are displayed using the last font in the font list as long as the list is too short.

The font list can also be changed by editing the text, if you copy in text from other buffers that uses other fonts, or if you specify a font by name. If such activity requires the use of a font not already in the font list, it is added at the end of the list, and you must say whether to make the change permanent in the file’s **Fonts** attribute.

You can print a list of all available font names with **M-X List Fonts**. The font names are mouse-sensitive; use **[L]** on a name to print a sample of a font, which shows what all the characters in the font look like. The command **M-X Display Font** can also be used to print a sample of a font; use the minibuffer to specify the font name.

23.2.2 Specifying a Font from the List

Once you have specified the list of fonts to work with, each time you use a font-change command you must specify one of the fonts from the list.

The simplest form of font specification is a letter: ‘**A**’ for the first font in the list, ‘**B**’ for the second, and so on. This assumes you remember the order of the fonts in the list, which normally you will. For example, **M-J B** means to convert the word after point to font B, the second font in the list.

You can also click **[L]** on any character of text in the same buffer, to specify that character’s font. **[R]** presents a menu with an entry for each of the fonts in the list.

The most general way to specify a font is to type **(ALTMODE)**, which brings up a minibuffer in which you can specify a font by name. If the font is not already in the font list, it is added at the

end; when this happens, you are asked whether to modify the file's **Fonts** attribute to include the new font.

23.2.3 Font-Change Commands

C-J *f* Change the character after point to font *f*.

M-J *f* Change the word after point to font *f*.

C-M-J *f* Specify the font for newly inserted text to *f*.

C-X C-J *f*

C-Shift-J *f*

Change the characters in the region to font *f*.

M-Shift-J *f g*

Change all characters in the region that are presently in font *f* into font *g*.

M-X Electric Font Lock Mode

Enable or disable automatic insertion of LISP comments in font B.

At any time when a multi-font buffer is selected, ZMACS identifies one font as the *current font*. Its font-letter and name appear in the mode line. This font is used for all text inserted in the buffer, except for text that carries its own font information. In particular, it is used for text that you type in.

To select a current font, use the command **C-M-J *f* (Change Default Font)**, where *f* is a font specification.

The simplest commands for changing the font of text already in the buffer are **C-J *f* (Change Font Char)** and **M-J *f* (Change Font Word)**. These move point by characters or words, changing all the characters moved over into font *f*. If these commands are given several times in succession (nothing but numeric arguments intervening), only the first one reads a font specifier *f*. The following commands use the same font as the first one.

A more general command to change the font of text is **Change Font Region**, which you can invoke as **C-Shift-J *f*** or **C-X C-J *f***.

You can also change the font of selected characters with **M-Shift-J *f g* (Change One Font Region)**. In this command you specify two fonts; then, throughout the region, each character in the first font is changed to the second. Characters in other fonts are not affected. You could use this, for example, to change all the italic characters in one paragraph into bold face.

Electric Font Lock mode is a minor mode in which text that you type in goes in font B when inside a LISP comment, and in font A otherwise. When this mode is in use, you cannot insert text in any font except A and B, though you can change text already in the buffer to other fonts. Use the command **M-X Electric Font Lock Mode** to turn this mode on or off. See section 31.1 [Minor Modes], page 191.

23.2.4 Fonts and Copying Text

When you copy text from one multi-font buffer to another, each character's font is preserved by *name*. If a character was in font TR10 in the original buffer, it remains in font TR10 in the buffer it is copied into. This can require adding the font TR10 to the font list of that buffer.

When text is copied from a fontless buffer into a multi-font buffer, it is put into the current insertion font.

When text is copied into a fontless buffer, its font information is discarded.



24. Commands for Natural Languages

The term *text* has two widespread meanings in our area of the computer field. One is, data that is a sequence of characters. Any file that you edit with ZMACS is *text*, in this sense of the word. The other meaning is more restrictive; it is, a sequence of characters in a human language for humans to read (possibly after processing by a text formatter), as opposed to a program or commands for a program.

Human languages have syntactic/stylistic conventions that can be supported or used to advantage by editor commands: conventions involving words, sentences, paragraphs, and capital letters. This chapter describes ZMACS commands for all of these things. In addition, there is a major mode, Text mode, which customizes ZMACS in small ways for editing a file of human language text. There are also commands for *filling*, or rearranging paragraphs into lines of approximately equal length, and for automatic manipulation of text formatter commands.

The commands for moving over and killing words (see section 24.2 [Words], page 126), sentences (see section 24.3 [Sentences], page 127) and paragraphs (see section 24.4 [Paragraphs], page 128) are primarily intended for natural-language text, but are very often useful in editing programs also.

24.1 Text Mode

Editing files of text in a human language ought to be done using Text mode rather than LISP or Fundamental mode. Invoke **M-X Text Mode** to enter Text mode. In Text mode, **Tab** runs the function **Tab To Tab Stop**, which allows you to use arbitrary tab stops set with **M-X Edit Tab Stops** (see section 22.1 [Tab Stops], page 117). Features concerned with comments in programs are turned off except when explicitly invoked. The syntax table is changed so that periods are not considered part of a word, while apostrophes, backspaces and underlines are.

Text mode is sometimes used on machine-parsable data bases whose syntactic conventions are enough like those of English for it to be useful. But often these data bases are unlike actual text in that the division of the data base into lines has semantic significance; filling the data base would make it incorrect. Since many users like to turn on Auto Fill mode automatically whenever they use Text mode, it is vital to be able to stop them from doing this on data bases that can't handle it. This is done with a **Nofill** attribute (see chapter 30 [Attributes], page 187). Its function is to tell **zwei:auto-fill-if-appropriate** that Auto Fill mode should not be used (see section 24.6 [Filling], page 130).

24.2 Words

ZMACS has commands for moving over or operating on words. By convention, they are all **Meta-** characters.

- M-F** Move Forward over a word.
- M-B** Move Backward over a word.
- M-D** Kill up to the end of a word.
- M-RUBOUT** Kill back to the beginning of a word.
- M-@** Mark the end of the next word.
- M-T** Transpose two words; drag a word forward or backward across other words.

Notice how these commands form a group that parallels the character based commands **C-F**, **C-B**, **C-D**, **C-T** and **RUBOUT**. **M-@** is related to **C-@**, which is an alias for **C-SPACE**.

The commands **Meta-F (Forward Word)** and **Meta-B (Backward Word)** move forward and backward over words. They are thus analogous to **Control-F** and **Control-B**, which move over single characters. Like their **Control-** analogues, **Meta-F** and **Meta-B** move several words if given an argument. **Meta-F** with a negative argument moves backward, and **Meta-B** with a negative argument moves forward. Forward motion stops right after the last letter of the word, while backward motion stops right before the first letter.

Meta-D (Forward Kill Word) kills the word after point. To be precise, it kills everything from point to the place **Meta-F** would move to. Thus, if point is in the middle of a word, **Meta-D** kills just the part after point. If some punctuation comes between point and the next word, it is killed along with the word. If you wish to kill only the next word but not the punctuation before it, simply do **Meta-F** to get the end, and kill the word backwards with **Meta-RUBOUT**. **Meta-D** takes arguments just like **Meta-F**.

Meta-RUBOUT (Backward Kill Word) kills the word before point. It kills everything from point back to where **Meta-B** would move to. If point is after the space in 'FOO, BAR', then 'FOO, ' is killed. If you wish to kill just 'FOO', do **Meta-B Meta-D** instead of **Meta-RUBOUT**.

Meta-T (Exchange Words) moves the cursor forward over a word, dragging the word preceding or containing the cursor forward as well. A numeric argument serves as a repeat count. **Meta-T** with a negative argument has the opposite effect of **Meta-T** with a positive argument; it drags the word behind the cursor backward over a word. An argument of zero is special: **M-0 M-T** exchanges

the word at point (surrounding or adjacent to it) with the word at mark. In any case, the delimiter characters between the words do not move. For example, 'FOO, BAR' exchanges into 'BAR, FOO' rather than 'BAR FOO,'.

To operate on the next n words with an operation that applies between point and mark, you can either set the mark at point and then move over the words, or you can use the command **Meta-C** (**Mark Word**) which does not move point, but sets the mark where **Meta-F** would move to. It can be given arguments just like **Meta-F**.

Note that if you are in Atom Word mode and in LISP mode, all the word commands regard an entire LISP atom as a single word. See section 31.1 [Minor Modes], page 191.

The word commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to be a word delimiter. See section 31.5 [Syntax], page 201.

24.3 Sentences

The ZMACS commands for manipulating sentences and paragraphs are mostly **Meta-** commands, so as to parallel the word-handling commands.

M-A Move back to the beginning of the sentence.

M-E Move forward to the end of the sentence.

M-K Kill forward to the end of the sentence.

C-X **RUBOUT**

Kill back to the beginning of the sentence.

The commands **Meta-A** and **Meta-E** (**Backward Sentence** and **Forward Sentence**) move to the beginning and end of the current sentence, respectively. They were chosen to resemble **Control-A** and **Control-E**, which move to the beginning and end of a line. Unlike them, **Meta-A** and **Meta-E** if repeated or given numeric arguments move over successive sentences. ZMACS considers a sentence to end wherever there is a '.', '?' or '!' followed by the end of a line or two spaces, with any number of ')', ']', ',', or '*' characters allowed in between. A sentence also begins or ends wherever a paragraph begins or ends.

Neither **M-A** nor **M-E** moves past the newline or spaces beyond the sentence edge at which it is stopping.

Just as **C-A** and **C-E** have a kill command, **C-K**, to go with them, so **M-A** and **M-E** have a corresponding kill command **M-K** (**Kill Sentence**) which kills from point to the end of the sentence. With minus one as an argument it kills back to the beginning of the sentence. Larger arguments serve as a repeat count.

There is a special command, **C-X** **RUBOUT** (**Backward Kill Sentence**) for killing back to the beginning of a sentence, because this is useful when you change your mind in the middle of composing text. This is the analog for sentences of **CLEAR-INPUT** for lines.

24.4 Paragraphs

The ZMACS commands for manipulating paragraphs are also **Meta-** commands.

- M-[** Move back to previous paragraph beginning.
- M-]** Move forward to next paragraph end.
- M-H** Put point and mark around this or next paragraph.

Meta-[(**Backward Paragraph**) moves to the beginning of the current or previous paragraph, while **Meta-]** (**Forward Paragraph**) moves to the end of the current or next paragraph. Blank lines and text formatter command lines separate paragraphs and are not part of any paragraph. Also, an indented line starts a new paragraph.

In major modes for programs (as opposed to Text mode), paragraphs begin and end only at blank lines. This makes the paragraph commands continue to be useful even though there are no paragraphs per se.

When there is a fill prefix, then paragraphs are delimited by all lines that don't start with the fill prefix. See section 24.6 [Filling], page 130.

When you wish to operate on a paragraph, you can use the command **Meta-H** (**Mark Paragraph**) to set the region around it. This command puts point at the beginning and mark at the end of the paragraph point was in. If point is between paragraphs (in a run of blank lines, or at a boundary), the paragraph following point is surrounded by point and mark. If there are blank lines preceding the first line of the paragraph, one of these blank lines is included in the region.

Thus, for example, **M-H C-W** kills the paragraph around or after point.

The precise definition of a paragraph boundary is controlled by the variables `zwei:*paragraph-delimiter-list*`, `zwei:*text-justifier-escape-list*` and `zwei:*page-delimiter-list*`. The value of each of these variables is a list of characters. The first character of a line is tested for membership in each of the three lists.

If the first character is a member of `zwei:*paragraph-delimiter-list*`, then the line either separates paragraphs (if that character is also a member of `zwei:*text-justifier-escape-list*`) or is the first line of a paragraph. For example, '␣' is a member of both lists, so that lines starting with '␣' separate paragraphs, but `(SPACE)` is a member of `zwei:*paragraph-delimiter-list*` only, so that lines starting with `(SPACE)` start new paragraphs and are part of those paragraphs.

If the first character of a line is a member of `zwei:*page-delimiter-list*`, the line is a page delimiter line, and is treated also as separating paragraphs just as a blank line does. See section 24.5 [Pages], page 129.

24.5 Pages

Files are often thought of as divided into *pages* by the character `(PAGE)`. For example, if a file is printed on a line printer, each page of the file, in this sense, will start on a new page of paper. Many editors make the division of a file into pages extremely important. For example, they may be unable to show more than one page of the file at any time. ZMACS treats a `(PAGE)` character just like any other character. It can be inserted with `C-Q (CLEAR-SCREEN)` (because `(CLEAR-SCREEN)` is the keyboard name for `(PAGE)`), or deleted with `(RUBOUT)`. Thus, you are free to paginate your file, or not. However, since pages are often meaningful divisions of the file, commands are provided to move over them and operate on them.

- C-X C-P** Put point and mark around this page (or another page).
- C-X [** Move point to previous page boundary.
- C-X]** Move point to next page boundary.
- C-X L** Count the lines in this page.

The **C-X [(Previous Page)** command moves point to immediately after the previous page delimiter. If point is already right after a page delimiter, it skips that one and stops at the previous one. A numeric argument serves as a repeat count. The **C-X] (Next Page)** command moves forward past the next page delimiter.

The **C-X C-P** command (**Mark Page**) puts point at the beginning of the current page and the mark at the end. The page delimiter at the end is included (the mark follows it). The page

delimiter at the front is excluded (point follows it). This command can be followed by **C-W** to kill a page that is to be moved elsewhere. If it is inserted after a page delimiter, at a place where **C-X]** or **C-X [** would take you, then the page will be properly delimited before and after once again.

A numeric argument to **C-X C-P** is used to specify which page to go to, relative to the current one. Zero means the current page. One means the next page, and -1 means the previous one.

The **C-X L** command (**Count Lines Page**) is good for deciding where to break a page in two. It prints in the echo area the total number of lines in the current page, and then divides it up into those preceding the current line and those following, as in

Page has 96 (72+25) lines

Notice that the sum is off by one; this is correct if point is not at the front of a line.

A line is considered a page separator if its first character is a member of the list **zwei:*page-delimiter-list***. The next page begins just after the first character on the line. Page separator lines are also considered paragraph separators.

24.6 Filling Text

With Auto Fill mode, text can be *filled* (broken up into lines that fit in a specified width) as you insert it. If you alter existing text it may no longer be properly filled; then explicit commands for filling can be used.

M-X Auto Fill Mode

Enable or disable Auto Fill mode.

SPACE

RETURN

- . ? ! In Auto Fill mode, break lines when appropriate.
- M-Q** Fill paragraph.
- M-G** Fill region as one paragraph.
- M-S** Center a line.

Entering Auto Fill mode is done with **M-X Auto Fill Mode**. You can see that Auto Fill mode is in effect by the presence of the word 'Fill' in the mode line, inside the parentheses. Auto

Fill mode is a minor mode, turned on or off for each buffer individually. See section 31.1 [Minor Modes], page 191.

In Auto Fill mode, lines are broken automatically at spaces when they get longer than the desired width. Inserting characters into the middle of a line can move text from the end of that line into the following line. Line breaking and rearrangement takes place only when you type **SPACE**, **RETURN**, or one of `!.?'`. If you wish to insert one of these characters without permitting line-breaking, quote it with **C-Q**. Alternatively, **C-O** inserts a newline without line breaking.

Auto Fill mode works well with LISP mode, because when it makes a new line in LISP mode it indents that line with **TAB**. If a line ending in a comment gets too long, the text of the comment is split into two comments.

The set of characters that trigger line-breaking in Auto Fill mode is controlled by the variable `zwei:auto-fill-activation-characters*`, whose value is a list of characters.

Auto Fill mode does not refill entire paragraphs, however, just one or two lines. So editing in the middle of a paragraph can result in a paragraph that is not correctly filled. To refill a paragraph, use the command **Meta-Q (Fill Paragraph)**. It causes the paragraph that point is inside, or the one after point if point is between paragraphs, to be refilled. All the line-breaks are removed, and then new ones are inserted where necessary. **M-Q** can be undone with **C-Shift-U**. See chapter 12 [Undo], page 53.

Meta-Q uses the same criteria as **Meta-H** for finding the bounds of a paragraph, and always fills one complete paragraph. See section 24.4 [Paragraphs], page 128. For more control, you can use **Meta-G (Fill Region)** which refills everything between point and mark. **Meta-G** recognizes only blank lines as paragraph separators.

Paragraph filling inserts double spaces after certain characters when they appear at the end of a line. The variable `zwei:fill-extra-space-list*` controls this; its value is a list of characters that warrant an extra space. By default, they are `!.?'`.

A numeric argument to **M-G** or **M-Q** causes it to *justify* the text as well as filling it. This means that extra spaces are inserted to make the right margin line up exactly at the fill column. Extra spaces are removed by **M-Q** or **M-G** with no argument.

The command **Meta-S (Center Line)** centers the current line within the current fill column. With an argument, it centers several lines individually and moves past them.

The maximum line width for filling is in the variable `zwei:fill-column*`. Its value is actually in pixels, not characters. This is because ZMACS can use variable-width fonts, and calculates line widths for filling using the actual widths of the characters on the line. The easiest way to set `zwei:fill-column*` is to use the command `C-X F` (**Set Fill Column**). With no argument, it sets `zwei:fill-column*` to the current horizontal position of point. With a numeric argument, it uses that as the new fill column. If it is greater than 200, it is regarded as a number of pixels, otherwise as a number of characters (to be multiplied by the width of a space in font A).

To fill a paragraph in which each line starts with a special marker (which might be a few spaces, giving an indented paragraph), use the *fill prefix* feature. The fill prefix is a string that ZMACS expects every line to start with, and that is not included in filling. It is stored in the variable `zwei:fill-prefix*`.

To specify a fill prefix, move to a line that starts with the desired prefix, put point at the end of the prefix, and give the command `C-X .` (**Set Fill Prefix**). That's a period after the `C-X`. To turn off the fill prefix, specify an empty prefix: type `C-X .` with point at the beginning of a line.

When a fill prefix is in effect, the fill commands remove the fill prefix from each line before filling and insert it on each line after filling. In Auto Fill mode, `(SPACE)` also inserts the fill prefix on any new line. Lines that do not start with the fill prefix are considered to start paragraphs, both in `M-Q` and the paragraph commands; this is just right if you are using paragraphs with hanging indentation (every line indented except the first one). Lines that are blank or indented once the prefix is removed also separate or start paragraphs; this is what you want if you are writing multi-paragraph comments with a comment delimiter on each line.

A special command is provided for filling paragraphs within multi-line comments: `M-X Fill Long Comment`. It applies to the line point is on and all comment lines consecutive with it. All the comment delimiters are removed, the text is filled as one or more paragraphs according to its contents, and comment delimiters are put back on each line. A generalized comment delimiter for LISP code can include any number of semicolons followed by any number of spaces. The lines to be filled are compared to find the longest generalized comment delimiter that all the lines share; this is then used as a temporary fill prefix.

Many users like Auto Fill mode and want to use it in all text files. Execute the following LISP expression, perhaps in your init file, to cause Auto Fill mode to be turned on whenever Text mode is entered:

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

There is one exception: Auto Fill will not be turned on if the file has a **Nofill** attribute. Give such attributes to files that use Text mode but should not be filled. See chapter 30 [Attributes], page 187.

24.7 Editing Text Formatter Directives

ZMACS has commands for manipulating text-formatter font-change directives. These commands do not produce any sort of font changes in the text file itself; they insert or move characters in the text that direct text formatters to output font changes.

- M-#** Change text formatter output font for previous word, or next word.
- C-X #** Change text formatter output font for the region.

24.7.1 Font Change Commands

The commands for text formatter font changes work with the text formatters R, Bolio and BoTeX. A font change is assumed to be of the form '*edigit*', meaning select font *digit*, or '*ε**', meaning return to the previously selected font. '*ε*' is ASCII **CTRL-F**.

M-# (**Change Font Word**) is a command to change the font of a word. Its action is rather complicated to describe, but that is because it tries to be as versatile and convenient as possible in practice.

If you type **M-#** with an argument, the previous word is put into the font specified by the argument. Point is not changed. This means that, if you want to insert a word and put it in a specific font, you can type the word, then use **M-#** to change its font, and then go on inserting. The font is changed by putting '*edigit*' before the word and a '*ε**' after.

If you type **M-#** with no argument, it takes the last font change (either '*edigit*' or '*ε**', whichever is closer) and moves it one word forward. What this implies is that you can change the font of several consecutive words incrementally by moving after the first word, issuing **M-#** with an argument to set that word's font, and then typing **M-#** to extend the font change past more words. Each **M-#** advances past one more word.

M-# with a negative argument is the opposite of **M-#** with no argument; it moves the last font change *back* one word. If you type too many **M-#**'s, you can undo the extra ones this way. If you

move one font change past another, one or both are eliminated, so as to do the right thing. As a result, **M-Minus M-#** cancels out the effect of a **M-#** with an argument. Try it!

You can also change the font of a whole region by putting point and the mark around it and issuing **C-X # (Text Formatter Change Font Region)**, with the font number as argument. **C-X #** with a negative argument removes all font changes inside or adjacent to the region.

25. Editing LISP Code

ZMACS has many commands designed to understand the syntax or even the semantics of LISP, to speed editing of LISP programs. These commands can

- Move over or kill balanced expressions (see section 25.2 [Lists], page 136).

- Move over or kill top-level balanced expressions, known as *defuns* (see section 25.3 [Defuns], page 138).

- Show how parentheses balance (see section 25.5 [Matching], page 143).

- Insert, kill or align comments (see section 25.6 [Comments], page 143).

- Follow the usual conventions for indentation of LISP code (see section 25.4 [LISP Indentation], page 139).

- Follow popular conventions for use of upper and lower case letters in LISP code (see <undefined> [LISP Case], page <undefined>).

- Display the documentation of the LISP functions called by the program and global variables it uses (see <undefined> [LISP Documentation], page <undefined>).

- Transform certain LISP constructs into other semantically equivalent ones (see <undefined> [LISP Semantics], page <undefined>).

The commands for words, sentences and paragraphs are very useful in editing LISP code even though their canonical application is for editing natural language text. Most LISP symbols contain words; sentences can be found in strings and comments. Paragraphs per se are not present in LISP code, but the paragraph commands are useful anyway, because LISP mode defines paragraphs to begin and end at blank lines. Judicious use of blank lines to make the program clearer will also provide interesting chunks of text for the paragraph commands to work on. See chapter 24 [Text], page 125.

25.1 LISP Mode

LISP programs should be edited in LISP mode. In this mode, **(TAB)** is defined to indent the current line according to the conventions of LISP programming style. It does not matter where in the line **(TAB)** is used; the effect on the line is the same. The function that does the work is called **Indent For LISP**. **(LINEFEED)**, as usual, does a **(RETURN)** and a **(TAB)**, so it moves to the next line and indents it.

As in most modes where indentation is likely to vary from line to line, **(RUBOUT)** is redefined to treat a tab as if it were the equivalent number of spaces (the command function **Tab Hacking**

Rubout). This makes it possible to rub out indentation one column at a time without worrying whether it is made up of spaces or tabs. **Control-RUBOUT** does the ordinary type of rubbing out, which rubs out a whole tab at once.

Paragraphs are defined to start only with blank lines so that the paragraph commands can be useful. Auto Fill mode, if enabled in LISP mode, indents the new lines that it creates. Comments start with ';'. If Atom Word mode is in effect, then in LISP mode the word-motion commands regard each LISP atom as one word.

25.2 S-expressions and Lists

- C-M-F** Move Forward over s-expression.
- C-M-B** Move Backward.
- C-M-K** Kill s-expression forward.
- C-M-RUBOUT**
Kill s-expression backward.
- C-M-U** Move Up and backward in list structure.
- C-M-(** The same.
- C-M-)** Move up and forward in list structure.
- C-M-D** Move Down and forward in list structure.
- C-M-N** Move forward over a list.
- C-M-P** Move backward over a list.
- C-M-T** Transpose s-expressions.
- C-M-Q** Put mark after s-expression.

By convention, ZMACS commands that deal with balanced parentheses are usually **Control-Meta-** characters. They tend to be analogous in function to their **Control-** and **Meta-** equivalents. These commands are usually thought of as pertaining to LISP, but can be useful with any language in which some sort of parentheses exist (including English).

To move forward over an s-expression, use **C-M-F (Forward Sexp)**. If the first significant character after point is an '(', **C-M-F** moves past the matching ')'. If the character begins an atom, **C-M-F** moves to the end of the atom. If the first character is a ')', **C-M-F** just moves past it. (This last is not really moving across an s-expression; it is an exception that is included in the definition of **C-M-F** because it is as useful a behavior as anyone can think of for that situation.)

The command **C-M-B (Backward Sexp)** moves backward over an s-expression; it is like **C-M-F** with the argument negated. If there are any prefix characters (singlequote, backquote and comma, in LISP) preceding the s-expression, **C-M-B** moves back over them as well.

C-M-F or **C-M-B** with an argument repeats that operation the specified number of times; with a negative argument, it moves in the opposite direction.

The s-expression commands ignore comments completely if used when point is not inside a comment. Within a comment, they parse the text of the comment as if it were LISP code. In order to tell properly where comments start, they do all parsing forward and always start from the beginning of the defun. Moving backward is done by parsing forward until the starting point, meanwhile always remembering the beginning of the last s-expression that finished.

The s-expression commands can also tell whether point is inside a string. If it is, they parse the text inside the string as if it were LISP code, but refuse to move outside the string. From outside, the string counts as a single s-expression.

The *list commands* move over lists like the s-expression commands but skip blithely over any number of other kinds of s-expressions (symbols, strings, etc). They are **C-M-N (Forward List)** and **C-M-P (Backward List)**.

Killing an s-expression at a time can be done with **C-M-K** and **C-M-RUBOUT** (**Kill Sexp** and **Backward Kill Sexp**). **C-M-K** kills the characters that **C-M-F** would move over, and **C-M-RUBOUT** kills what **C-M-B** would move over.

C-M-F and **C-M-B** stay at the same level in parentheses, when that's possible. To move up one (or *n*) levels, use **C-M-(** or **C-M-)** (**Backward Up List** and **Forward Up List**). **C-M-(** moves backward up past one containing '('. **C-M-)** moves forward up past one containing ')'. If used when point is inside a string, these commands move out of the string and no farther. A positive argument serves as a repeat count; a negative argument reverses direction of motion and also requests repetition. **C-M-U** is another name for **C-M-(**.

To move down in list structure, use **C-M-D (Down List)**. It is nearly the same as searching for a '('.

A somewhat random-sounding command that is nevertheless easy to use is **C-M-T (Exchange Sexps)**, which drags the previous s-expression across the next one. An argument serves as a repeat count, and a negative argument drags backwards (thus canceling out the effect of **C-M-T** with a positive argument). An argument of zero, rather than doing nothing, transposes the s-expressions

at the point and the mark.

To make the region be the next s-expression in the buffer, use **C-M-@ (Mark Sexp)** which sets mark at the same place that **C-M-F** would move to. **C-M-@** takes arguments like **C-M-F**. In particular, a negative argument is useful for putting the mark at the beginning of the previous s-expression.

The list and s-expression commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to act like an open parenthesis. See section 31.5 [Syntax], page 201.

25.3 Defuns

C-M-[, **C-M-A**

Move to beginning of defun.

C-M-], **C-M-E**

Move to end of defun.

C-M-H Put region around whole defun.

In ZMACS, a list at the top level in the buffer is called a *defun*. The name derives from the fact that most top level lists in a LISP file are instances of the special form **defun**, but any top level list counts as a defun in ZMACS parlance regardless of what function it calls.

The commands to move to the beginning and end of the current defun are **C-M-[(Beginning Of Defun)** and **C-M-] (End Of Defun)**. Alternate names for these two commands are **C-M-A** for **C-M-[** and **C-M-E** for **C-M-]**.

If you wish to operate on the current defun, use **C-M-H (Mark Defun)** which puts point at the beginning and mark at the end of the current or next defun. For example, this is the easiest way to get ready to move the defun to a different place in the text.

ZMACS assumes that any open-parenthesis found in the leftmost column is the start of a defun. The LISP Machine compiler usually makes the same assumption in order to detect parenthesis errors early. Therefore: **never put an open-parenthesis at the left margin in a LISP file unless it is the start of a top level list**. If, for example, you want to have an open-parenthesis at the left margin inside a string, put an escape character (backslash in Common LISP, slash in traditional syntax) in front of it; this has no effect on the contents of the string but it keeps the open-parenthesis away from column zero.

In the remotest past, the original Emacs found defuns by moving upward a level of parentheses until there were no more levels to go up. This always required scanning all the way back to the beginning of the buffer, even for a small function. To speed up the operation, Emacs was changed to assume that any ' (' (or other character assigned the syntactic class of opening-delimiter) at the left margin is the start of a defun. This heuristic is nearly always right and avoids the costly scan. ZMACS uses the same convention.

25.4 LISP Indentation

The best way to keep LISP code properly indented ("ground") is to use ZMACS to re-indent it when it is changed. ZMACS has commands to indent properly either a single line, a specified number of lines, or all of the lines inside a single s-expression.

C-M-(TAB) Re-indent line according to parenthesis depth.

(TAB) In LISP mode, same as C-M-(TAB).

(LINEFEED)

Equivalent to **(RETURN)** followed by **(TAB)**.

C-(TAB) Indent current line to a different plausible place.

C-M-Q Re-indent all the lines within one list.

C-M- Re-indent all lines in the region.

M-X Stack List Vertically

Indent list after point, one element per line.

The basic indentation function is **indent for LISP**, which gives the current line the correct indentation as determined from the previous lines' indentation and parenthesis structure. This function is normally found on **C-M-(TAB)**, but when in LISP mode it is placed on **(TAB)** as well (Use **Meta-(TAB)** or **C-Q (TAB)** to insert a tab). If executed at the beginning of a line, it leaves point after the indentation; when given inside the text on the line, it leaves point fixed with respect to the characters around it.

When entering a large amount of new code, use **(LINEFEED)** (**Indent New Line**), which is equivalent to a **(RETURN)** followed by a **(TAB)**. In LISP mode, a **(LINEFEED)** creates or moves down onto a blank line, and then gives it the appropriate indentation.

C-(TAB) (**Indent Differently**) is useful for lines where the canonical LISP indentation is undesirable, perhaps because it fails to bring out the semantics most clearly. This command chooses a different plausible place in the previous line to indent under. Each time it is used, it tries the next

possibility, in a cyclic order, of all the plausible ones.

(TAB) indents the second and following lines of the body of an s-expression under the first line of the body; therefore, if you alter the indentation of one of the lines yourself, then **(TAB)** will indent successive lines of the same list to be underneath it. This is the right thing for functions that **(TAB)** indents unaesthetically.

When you wish to re-indent code that has been altered or moved to a different level in the list structure, you have several commands available. You can re-indent a specific number of lines by giving the ordinary indent command (**(TAB)**, in LISP mode) an argument. This indents as many lines as you say and moves to the line following them. Thus, if you underestimate, you can repeat command to indent additional lines.

You can re-indent the contents of a single s-expression by positioning point before the beginning of it and typing **Control-Meta-Q (Indent Sexp)**. The line the s-expression starts on is not re-indented; thus, only the relative indentation within the s-expression, and not its position, is changed. To correct the position as well, type a **(TAB)** before the **C-M-Q**.

Another way to specify the range to be re-indented is with point and mark. The command **C-M-\ (Indent Region)** applies **(TAB)** to every line whose first character is between point and mark. In LISP mode, this does a LISP indent.

M-X Stack List Vertically operates on the list that starts after point. It inserts newlines between the elements of the list, then indents all the lines of the list the way **C-M-Q** would do.

25.4.1 Customizing LISP Indentation

The indentation pattern for a LISP expression can depend on the function called by the expression. For each LISP function, you can choose among several predefined patterns of indentation, or define an arbitrary one with a LISP program.

The standard pattern of indentation is as follows: the second line of the expression is indented under the first argument, if that is on the same line as the beginning of the expression; otherwise, the second line is indented **zwei:*lisp-indent-lone-function-offset*** columns beyond the beginning of the function name. Each following line is indented under the previous line whose nesting depth is the same.

If the variable **zwei:*lisp-indent-offset*** is non-nil, it overrides the usual indentation pattern for

the second line of an expression. Such lines are always indented `zwei:*lisp-indent-offset*` more columns than the containing list.

If a list's first element is not thought to be a function, either because the first element is peculiar or because its context suggests that it is a clause in a `cond` or similar construct, its second line is indented right under the beginning of the first element.

The standard pattern is overridden for certain functions. Functions whose names start with `def` always indent the second line by `zwei:*lisp-defun-indentation*` extra columns beyond the beginning of the function name.

An even more specific override mechanism is `zwei:*lisp-indent-offset-alist*`. This is an alist whose elements map function names (symbols) into *offset-specs*; each element has the form

```
(function . offset-spec)
```

An *offset-spec* is usually a list of the form

```
(arg-number1 offset1 arg-number2 offset2 ...)
```

This means that a line starting with argument number *arg-number1* should be indented *offset1* spaces beyond the where the function name starts. Lines starting with later arguments are treated the same way, until the argument number gets large enough for the next *arg-number* to apply. Arguments are numbered starting at zero. For example, the element for `multiple-value-bind` is

```
(multiple-value-bind 1 3 2 1)
```

which means that the argument zero (the list of variables) gets no special treatment, argument one is indented three spaces more than the function name, and arguments two and following are indented only one space more than the function name. (Argument zero is conventionally placed on the same line as the function name.) The resulting indentation style looks like this:

```
(multiple-value-bind (x y z)
  (intern string)
  (print x)
  (print y)
  z)
```

Any macro *m* whose argument list uses **&body** is automatically given an entry on **zwei:lisp-indent-offset-alist*** of the form

```
(m first-body-arg-number 1)
```

so that arguments before the body are indented using the standard indentation pattern for function calls, but the body arguments have a fixed indentation. Usually the result of this is to make the non-body arguments stand out.

offset-spec may also be a symbol that has a function definition. This symbol is called as follows to compute the indentation for *function*:

```
(funcall offset-spec defun-start-bp indented-line-bp
         last-unmatched-openparen-bp last-complete-sexp-start-bp
         width-of-space-in-pixels function)
```

offset-spec can return three values. Only one of them should be non-**nil**, as they are three ways to specify the same information. If the first value is non-**nil**, it should be a buffer pointer; this line is indented to start under the column of that pointer. If the second value is non-**nil**, it is the indentation to use, in spaces. If the third value is non-**nil**, it is the offset to use (beyond the column the function name starts in). This feature is used for **prog** and **tagbody**.

ZMACS knows that certain special forms (and macros), such as **let** and **cond**, have arguments that are not expressions. Two variables inform ZMACS of the names of these special forms. **zwei:cond-clause-superiors*** is a list of special forms whose arguments should be indented as **cond** clauses. **zwei:indent-not-function-superiors*** lists the constructs whose arguments fail to be expressions for other reasons. Its elements look like this:

function The first argument to *function* is not an expression.

function-name argnum

Argument number *argnum* of *function* is not an expression.

function-name argnum offset

Argument number *argnum* of *function* is not an expression, and the second line of that argument should be indented with offset *offset* within the argument.

function-name argnum1 offset1 argnum2 offset2 ...

Argument number *argnum1* of *function* is not an expression, and the second line of that argument should be indented with offset *offset1* within the argument. Likewise for *argnum2* and *offset2*. An *offset* may be **t**; then it says nothing about how to indent

within the argument.

function-name t offset

All arguments of *function* are not expressions, and the second line of each argument should be indented with *offset offset* within the argument.

25.5 Automatic Display Of Matching Parentheses

The ZMACS parenthesis-matching feature is designed to show automatically how parentheses match in the text. Whenever point is located after a right (closing) delimiter, the matching left (opening) delimiter blinks. Whenever point is located before a left delimiter, the matching right delimiter blinks.

Automatic matching is useful only when the other end of the list is on the screen. **C-)** (**Show List Start**) is useful for seeing the beginning of the list before point when that is not on the screen. It prints some of the text within the beginning of the list in the echo area.

This feature is available for languages other than LISP. It may apply to characters other than parentheses. For example, in C, braces and square brackets are used in the same fashion. ZMACS knows which characters to regard as matching delimiters based on the syntax table, which is set by the major mode. See section 31.5 [Syntax], page 201.

Two variables control parenthesis match display. **zwei:*flash-matching-paren*** turns the feature on or off; **nil** turns it off, but the default is **t** to turn match display on. **zwei:*flash-matching-paren-max-lines*** specifies how many lines to search to find the matching parenthesis. If the match is not found in that far, scanning stops.

The features described here help to indicate how parentheses balance locally. To localize parenthesis errors within a file, the first step is to use **M-X Find Unbalanced Parentheses**. See **<undefined>** [LISP Debug], page **<undefined>**.

25.6 Manipulating Comments

The comment commands insert, kill and align comments. There are also commands for moving through existing code and inserting comments.

M-; Insert or align comment.

- C-;** The same.
- C-M-;** Kill comment on current line. With region, kill comments in region.
- C-X ;** Set comment column.
- C-X C-;** Put a comment starter in front of each line of the region, or delete the comment starters.
- M-N** Move to Next line and insert comment.
- M-P** Move to Previous line and insert comment.
- M-LINEFEED**
The same.

The command that creates a comment is **Meta-;** or **Control-;** (**indent for Comment**). If there is no comment already on the line, a new comment is created, aligned at a specific column called the *comment column*. The comment is created by inserting whatever string ZMACS thinks should start comments in the current major mode. Point is left after the comment-starting string. If the text of the line goes past the comment column, then the indentation is done to a suitable boundary (usually, a multiple of 8 spaces in font A).

Meta-; can also be used to align an existing comment. If a line already contains the string that starts comments, then **M-;** just moves point after it and re-indents it to the right column. Exception: comments starting in column 0 are not moved.

Even when an existing comment is properly aligned, **M-;** is still useful for moving directly to the start of the comment.

C-M-; (**Kill Comment**) kills the comment on the current line, if there is one. The indentation before the start of the comment is killed as well. If there does not appear to be a comment in the line, nothing is done. To reinsert the comment on another line, move to the end of that line, do **C-Y**, and then do **M-;** to realign it.

To kill many comments, set up a region and then use **C-M-;**. It kills all the comments in the region.

25.6.1 Multiple Lines of Comments

If you wish to align a large number of comments, give **Meta-;** an argument, and it indents what comments exist on that many lines, creating none. Point is left after the last line processed (unlike the no-argument case).

When adding comments to a long stretch of existing code, the commands **M-N (Down Comment Line)** and **M-P (Up Comment Line)** may be useful. They are like **C-N** and **C-P** except that they do a **C-**; automatically on each line as you move to it, and delete any empty comment from the line as you leave it. Thus, you can use **M-N** to move down through the code, putting text into the comments when you want to, and allowing the comments that you don't fill in to be removed because they remained empty.

If you are typing a comment and find that you wish to continue it on another line, you can use the command **Meta-(LINEFEED) (Indent New Comment Line)**, which terminates the comment you are typing, creates or gobbles a new blank line, and begins a new comment indented under the old one. When Auto Fill mode is on, going past the fill column while typing a comment causes the comment to be continued in just this fashion. Note that if the next line is not blank, a blank line is created, and the continuation goes on that line. By comparison, **M-N** would create a continuation comment on the next existing line of code.

A special command is provided for filling paragraphs within multi-line comments: **M-X Fill Long Comment**. It applies to the line point is on and all comment lines consecutive with it. All the comment delimiters are removed, the text is filled as one or more paragraphs according to its contents, and comment delimiters are put back on each line. A generalized comment delimiter for LISP code can include any number of semicolons followed by any number of spaces. The lines to be filled are compared to find the longest generalized comment delimiter that all the lines share; this is then used as a temporary fill prefix. See section 24.6 [Filling], page 130.

25.6.2 Commenting Out Code

C-X C-; (**Comment Out Region**) is used for making non-comment text into comments, or vice versa. This is unlike the other comment commands, which attempt to have no effect on text outside of comments.

C-X C-; with no numeric argument inserts a comment start string at the beginning of each nonblank line that starts in the region. This includes lines that already start with comment start strings; they wind up with two (or one more than they had) comment start strings. **C-U C-X C-;** deletes one comment start string from the beginning of each line starting in the region. It cancels the effect of a prior **C-X C-;** with no argument. Other positive numeric arguments specify the number of copies of the comment starter to insert on each line; negative ones specify the number of copies to delete from each line.

Another way to comment out LISP code is to put it inside a **'#|...|#'** construct. Just insert

'#' in front of the unwanted code and '|#' after. '#|...|#' constructs may be nested.

25.6.3 Double and Triple Semicolons in LISP

In LISP code there are conventions for comments that start with more than one semicolon. Comments that start with two semicolons are indented as if they were lines of code, instead of at the comment column. Comments that start with three semicolons are supposed to start at the left margin. ZMACS understands these conventions by indenting a double-semicolon comment using **(TAB)**, and by not changing the indentation of a triple-semicolon comment at all. (Actually, this rule applies whenever the comment starter is a single character and is duplicated).

25.6.4 Options Controlling Comments

The comment column is stored in the variable **zwei:*comment-column***. You can set it to a number explicitly; note that the value is in pixels, not characters. Alternatively, the command **C-X ; (Set Comment Column)** sets the comment column to the column point is at. **C-U C-X ;** sets the comment column to match the last comment before point in the buffer, and then does a **Meta-** to align the current line's comment under the previous one.

Many major modes supply default local values for the comment column. Otherwise, if you change the variable itself, it changes globally (for all buffers) unless it has been made local in the selected one. See section 31.2 [Variables], page 192.

The string recognized as the start of a comment is stored in the variable **zwei:*comment-start***, while the string used to start a new comment is kept in **zwei:*comment-begin***. This makes it possible, for example, for you to have any ';' recognized as starting a comment but have new comments begin with '**; ****'.

In some languages, comments require a specific ending delimiter. The major modes for these languages set the variable **zwei:*comment-end*** to that string (normally it is **nil**). However, few commands do anything with the comment end string.

The value of **zwei:*comment-round-function*** is a LISP function that is used to compute where to align a comment on a line whose text extends past the comment column. This function is called with one argument, the horizontal position (in units of pixels) after the last non-blank non-comment text on the line, and it should return the horizontal position at which the comment should start (also in pixels).

25.7 Case Conventions for LISP Code

Some people like to write LISP symbols in upper case, while others write them in lower case. It makes no difference to the LISP system. But text in strings is case-sensitive, and people generally want the text in comments to follow the conventions of their own language regardless of the convention used for the code. ZMACS provides special facilities for converting the LISP code from upper case to lower, or vice versa, without affecting the case-sensitive parts of the text.

M-X Uppercase Lisp Code In Region

Convert region to upper case, except for strings, comments and quoted characters.

M-X Lowercase Lisp Code In Region

Convert region to lower case, except for strings, comments and quoted characters.

M-X Electric Shift Lock Mode

Enable or disable automatic insertion of LISP symbol names in upper case.

M-X Set Lowercase

Set the current buffer's **Lowercase** attribute.

A popular case convention for LISP code is to write all LISP symbols in upper case, but use mostly lower case for comments. Electric Shift Lock mode makes it automatic to insert text to follow this convention. Turn this mode on or off for the current buffer using the command **M-X Electric Shift Lock Mode**; the words 'Electric Shift-Lock' appear in the mode line when the mode is in effect. Electric Shift Lock mode is a minor mode; See section 31.1 [Minor Modes], page 191.

When Electric Shift Lock mode is turned on, you can type lower case text and it is converted automatically to upper case provided it is in a syntactic context where its case does not matter. Specifically, characters inside LISP strings and comments, and characters inside escape constructs (within vertical-bar groupings or following the single-character escape) are not converted. Outside of these contexts, each letter that you type is case-flipped: if you type it as lower case, it is inserted as upper case, and vice versa. (If the variable `zwei:*electric-shift-lock-xors*` is set to `nil`, all letters are converted to upper case.) Electric Shift Lock mode applies only to text that is inserted due to the input of self-inserting characters. It has no effect on text that is yanked, read from files, copied from registers, etc.

Here is an example of code typed using Electric Shift Lock mode, without ever pressing the **SHIFT** key:

```
(DEFUN SQUARE (X) ;function to do squaring
```

```
"return x times x"
(* X X)
```

If you customarily use this case convention, you may wish to have Electric Shift Lock mode in effect whenever you edit LISP code. You can do this by executing the LISP expression

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

in your init file.

If you prefer to write LISP code in lower case, you do not need any special assistance instead of Electric Shift Lock mode. However, you might wish to prevent other users from turning Electric Shift Lock mode on automatically when editing a file that is primarily lower case. To do this, give the file a **Lowercase** attribute. A non-nil **Lowercase** attribute tells **zwei:electric-shift-lock-if-appropriate** not to do anything for that file. The command **M-X Set Lowercase** can be used to set this attribute for the current buffer or the current file. See chapter 30 [Attributes], page 187.

There are special commands for converting LISP code from primarily upper case to primarily lower case. These change the case of LISP symbol names (in which case is ignored), but do not change letters inside strings, comments or vertical-bar groupings, or letters following slashes. The commands are **M-X Uppercase Lisp Code In Region** and **M-X Lowercase Lisp Code In Region**; both apply to the text in the region. Neither point nor mark should be inside a string, or the command will not function properly. They may be inside comments, though.

25.8 Editing Commands Based on LISP Semantics

The commands in this section transform LISP code based on the semantic relationships among certain important LISP special forms.

C-M-* Convert **cond** to **and**, **or** or **if**; or vice versa.

C-M-§ Convert old-style **do** to new-style or vice versa.

M-X Query Replace Let Binding

Replace variable with the value it is initialized to.

C-M-* (**Frob Lisp Conditional**) changes an **and** or an **or** or an **if** into a **cond** construct, or vice versa. The innermost applicable construct containing point is transformed.

After changing a **cond** into something else, point is left just before the close-parenthesis that terminates the construct. After changing to a **cond**, point is left at the end of the **cond** clause, just before the ')' that ends the **cond** clause. In such a situation, use **LINEFEED** to indent a new line for another form within the clause, or **M-)** to add a new **cond** clause. If an **or** or **and** with more than two arguments is changed to a **cond**, by default all but the last become part of a **cond**-condition, and only the last one becomes the consequent in the **cond**. A numeric argument to **C-M-#** specifies how many of the last arguments to put into the **cond**'s consequent.

C-M-# can transform a **cond** only if it has one clause, or two clauses where the second one starts with **t**. It refuses to act on other sorts of **cond**.

C-M-# (**Frob Do**) converts an (obsolete) old-style **do** into a new-style **do**. New-style **do**'s (at least eight years old by now) are the only kind legal in Common LISP. An old-style **do** looks like this:

```
(do var initform stepform endtest
  body...)
```

and its equivalent using new-style **do** is

```
(do ((var initform stepform))
    (endtest nil)
  body...)
```

M-X Query Replace Let Binding eliminates a **let**-variable by replacing it by its initial value. To use this command, first position point inside the **let** binding for the variable you want to eliminate. The binding itself is removed from the **let**, and then **Query Replace** is used to replace the variable by its initial value throughout the body of the **let**.

25.9 Editing Without Unbalanced Parentheses

M-(Put parentheses around next s-expression(s).

M-) Move past next close parenthesis and re-indent.

C-M-Shift-K

Delete innermost surrounding pair of matching parentheses.

C-M-Shift-F

Move innermost close parenthesis forward across s-expressions.

C-M-Shift-B

Move innermost open parenthesis backward across s-expressions.

The commands **M-(Make ()**) and **M-) (Move Over)** are designed to facilitate a style of editing that keeps parentheses balanced at all times. **M-(** inserts a pair of parentheses, either together as in '()', or, if given an argument, around the next several s-expressions, and leaves point after the open parenthesis. Instead of typing (F O O), you can type **M-(F O O**, which has the same effect except for leaving the cursor before the close parenthesis. Then you would type **M-)**, which moves past the close parenthesis, deleting any indentation preceding it (in this example there is none), and indenting with **(LINEFEED)** after it.

C-M-Shift-K (Delete ()) cancels the effect of **M-(** by deleting the innermost set of matching parentheses surrounding point. With argument *n*, it deletes the parentheses *n* levels out.

C-M-Shift-F (Grow List Forward) moves the close parenthesis of the list surrounding point, forward across *n* s-expressions, where *n* is the numeric argument (default 1). A negative numeric argument moves it backward across s-expressions. This has the effect of moving elements into the end of the list or out of it. To make the results more visible, the mark is set after the close parenthesis moved, *and activated* to make an underlined region.

C-M-Shift-F with argument *n* can always be canceled out by a repetition of the same command with argument *-n*.

C-M-Shift-B (Grow List Backward) works similarly but moves the open parenthesis that begins the list surrounding point. With a positive argument, it moves the open parenthesis backward, thus adding elements to the list.

M-X Reverse Following List reverses the order of the elements in the list that starts after point, by deleting them all and reinserting them. Thus, (a b (foo bar)) is changed to ((foo bar) b a). Point does not move. There may be an atom between point and the beginning of the list to be reversed.

25.10 Documentation Commands for LISP Code

As you edit LISP code in ZMACS, various commands are available for you to find out about the functions and variables you wish to use in the code.

C-Shift-A

Print argument names of function being called around point.

C-Shift-D

Print argument names and documentation string of function being called around point.

M-X Arglist

Print argument names of any function.

M-X Long Documentation

Print argument names and documentation of any function.

C-Shift-V

Print documentation and other info on variable name at point.

C-Shift-A (Quick Arglist) prints, in the echo area, the argument names and lambda-list keywords belonging to the function called in the innermost expression around point. This gives essentially the same information that the LISP function `arglist` would return as a list. If the function returns multiple values and contains a `values` declaration to name them, their names are printed as well, with an arrow to separate them from the arguments. For example, with point after `(intern ')`, **C-Shift-A** prints something like

```
INTERN: (SYM &OPTIONAL PKG) → (SYMBOL ALREADY-IN-FLAG PKG-FOUND-IN)
```

The command **C-Shift-D (Quick Documentation)** is similar but also prints the documentation string of the function being called. There is usually not room for this in the echo area, so both argument names and documentation appear as typeout, overlying part of the editing window.

If you wish to specify a function to see the argument names or documentation for, you can insert a call to the function into the buffer and then use **C-Shift-A** or **C-Shift-D**, or you can use one of the commands **M-X Arglist** or **M-X Long Documentation**. These commands use the minibuffer to read the name of the function to document. A function name can also be selected from a ZMACS window with the mouse, using \uparrow , while the mouse cursor is pointing straight up (which happens when the minibuffer is empty). **C-U C-Shift-D** is equivalent to **M-X Long Documentation**.

C-Shift-V (Describe Variable At Point) prints documentation about a variable. It first prints whether the symbol has a global value or is void, and then says whether the symbol is globally special, and, if so, which file contains the definition (`defvar` or alternative). If the variable has a documentation string, that is printed starting on the following line.



26. Running and Testing LISP Programs

The previous chapter discusses the ZMACS commands that are useful for making changes in LISP code. This chapter deals with commands that assist in the larger process of developing and maintaining LISP programs.

You can use ZMACS to compile and load your LISP program in order to test it (see section 26.2 [Compile File], page 157). You can also recompile individual functions after you change them (see section 26.3 [Compile Text], page 157). If compilation produces warnings, even compilation not done in ZMACS, special ZMACS commands can be used to find the functions that the warnings are about (see section 26.4 [Warnings], page 160). Other commands assist in debugging (see <undefined> [LISP Debug], page <undefined>).

ZMACS knows which lines of text correspond to each LISP function, so it can find the source code for any function specified by name. It also knows which functions you have changed the text of, and can recompile just the changed functions.

26.1 Sectionization

When ZMACS visits a file, it scans the text of the file line by line, dividing the file up into groups of lines that are semantic units. These groups of lines are called *sections*, and the process of finding them is called *sectionization*.

What constitutes a unit depends on the language that the file is written in (actually, on the major mode); in LISP code, a unit starts with a line that starts with an open-parenthesis, which means, if you follow the standard conventions for formatting LISP code, that each top-level LISP expression in the file is a unit. In major modes for editing input for text formatters, a unit begins with each text formatter command line that appears to define a section (or chapter, subsection, etc.) for the table of contents.

Every section is given a name. If possible, the name of a semantically meaningful object defined by the text of the section is used as the section name. This is done in LISP code when the LISP expression in the section is a function definition, variable definition, flavor definition, etc. If no name suggests itself directly from the text, a name is generated; it contains the file name, a number to make the section unique, and an indication of the kind of text in the section. In LISP mode, the name of the function called by the expression in the section is used for this.

Meta-. Move point to the definition of the specified function, variable, flavor, or anything else that is the name of a section.

M-X List Sections

Print list of all sections in a specified buffer.

M-X List Buffer Changed Sections

Print list of all changed sections in current buffer. Prepare to move to them, one by one, with **C-Shift-P**.

M-X Edit Buffer Changed Sections

Find all changed sections in current buffer. Move to the first one, and prepare to move to the others, one by one, with **C-Shift-P**.

M-X List Changed Sections

M-X Edit Changed Sections

Like the previous two commands but look through all buffers.

M-X Tags List Changed Sections

M-X Tags Edit Changed Sections

Like the previous two commands but look through all the buffers in the current buffer group.

M-X Tags Search List Sections

Look through all buffers in the current buffer group and print a list of all sections whose text contains a specified string.

M-X Find File No Sectionize

Find file but do not sectionize its buffer.

M-X Sectionize Buffer

Recalculate sections of current buffer from scratch.

Sections make possible the **Meta-.** command (**Edit Definition**), which moves point to the beginning of the definition of a specified LISP function, variable, flavor, etc. For example, **Meta-.** **read** **RETURN** finds and displays the definition of the LISP function **read**, while **Meta-.** **tv:window** **RETURN** finds and displays the definition of the flavor **tv:window**. Both system definitions and your own can be found using **Meta-.**

Meta-. reads an argument with the minibuffer, interpreting it as a symbol or function spec, and then finds the sections whose names match or suitably resemble the argument. It also looks for files not yet visited whose names appear in the **:source-file-name** property of the symbol or function spec; they are visited and sectionized, then their matching sections are found. These operations are not done all at once. The various leads are made into a possibilities list, and possibilities are tried until one matching section is found and displayed. Most likely this shows you the text you want. If it does not, use **Meta-.** with a numeric argument to see the next possibility. Repeating this will eventually show all the possible defining sections for the original argument.

Meta- pushes the previous location of point onto the point pdl so you can return there easily. See section 10.3 [Point Pdl], page 41.

There are special features to make it easier to specify the argument for **Meta-**, or any other ZMACS command whose argument is expected to be (usually, at least) the name of a LISP function.

1. A default argument is shown in the prompt; type just **(RETURN)** to use the default. The default is normally the name of the function called by the smallest LISP expression containing point.

In a few cases, a different default is chosen. For example, if the innermost expression calls **defun**, the function being defined is used as the default rather than **defun**.

2. The characters **(ALTMODE)**, **(SPACE)** and **(END)** perform completion on the argument. Note that only names of sections already known to ZMACS are available for completion. Names that could be found by visiting other files automatically are not known.
3. The mouse can be used to select an argument. If the minibuffer is empty, the mouse cursor has the shape of an arrow pointing straight up; this means that all function names in ZMACS buffers are mouse-sensitive. Clicking on one selects it as the argument.

When the minibuffer is not empty, the mouse has its usual cursor shape and its usual commands. These can be useful too; for example, they can be used to mark a region and copy it into the minibuffer.

A related command is **C-M-** (**Edit ZMACS Command**), which reads a ZMACS command and then finds the source code for the command function that it is connected to. For example, **C-M-** **M-F** visits the ZMACS source file **COMA.LISP** and moves point to the LISP function **com-forward-word**. **C-M-** **M-X Find File** visits the source file **ZMACS.LISP** and moves point to the LISP function **com-find-file**.

The simplest thing you can do with sections is list their names. **M-X List Sections** prints a list of all the section names of a buffer, whose name you must specify using the minibuffer. The names printed are mouse-sensitive; click **(L)** on one of them to move point to that section. This pushes the old value of point on the point pdl (see section 10.3 [Point Pdl], page 41).

You can print a list of sections whose text has been changed, using **M-X List Buffer Changed Sections**. This applies to the current buffer. The section names printed are mouse-sensitive. In addition, a list of all the changed sections is established as the current possibilities list. This means that the **C-Shift-P** command can be used to move point to the next one of these sections. See section 26.7 [Possibilities Lists], page 164. The command **M-X Edit Buffer Changed Sections** is like **M-X List Buffer Changed Sections** except that it moves point to the first of the sections

and does not print their names. You can find the rest of the changed sections with **C-Shift-P**.

The word “changed” is actually vague: changed since when? You can use a numeric argument to **M-X List Buffer Changed Sections** to say since when. No argument, or an argument of 1, selects sections changed at any time since the file was visited. This is the most inclusive criterion for “changed” sections. An argument of 2 says to list only sections changed since the last time the buffer was saved. This shows sections whose text does not match the latest version of the file. An argument of 3 says to list only sections changed since the last time their text was evaluated or compiled. This shows sections whose text does not match the definitions in the LISP environment.

Other similar commands apply to several buffers at once. **M-X Edit Changed Sections** and **M-X List Changed Sections** look in all buffers that have sections. **M-X Tags Edit Changed Sections** and **M-X Tags List Changed Sections** look in all buffers visiting files in the current buffer group. All four take numeric arguments to specify “changed since when”.

There are also commands to do things to the text of changed sections. You can add them to the current patch (see section 26.8 [Patches], page 165), evaluate them or compile them (see section 26.3 [Compile Text], page 157).

Note that if you make a change to the text in a section and then undo it with **C-Shift-U**, the section is still considered “changed”. It would be preferable to cancel the “changed” indication when the changes are undone, but it is hard to implement this.

Another way to find a list of interesting sections is with **M-X Tags Search List Sections**. It searches buffers in the current buffer group for sections whose text contains a specified string. See section 19.6 [Buffer Groups], page 104, for information on this.

In some cases, you may not want ZMACS to sectionize a file. For example, if you visit an old version of a LISP program, you might want **Meta-** to ignore it, and look in the current version instead for any functions defined in that file. Or the file may contain text that does not fit the usual conventions for its major mode, and would produce nonsensical sections. To prevent sectionization, use the command **M-X Find File No Sectionize** instead of **C-X C-F** to visit the file. If ZMACS strongly suspects that it should sectionize the file after all—for example, if you do **Meta-** on a function that is expected to be in that file—it will offer to sectionize the file then.

If ZMACS is wrong about the sections in a buffer, you can fix them with **M-X Sectionize Buffer**. The most likely reason ZMACS would be wrong is that you have changed the major mode. The criteria for where sections start and what their names are depend on the major mode, but selecting a new major mode in a buffer does not automatically recompute the buffer's sections.

You do not need to use **M-X Sectionize Buffer** just because you have changed the buffer's text. ZMACS automatically rescans changed lines in order to detect any changes in the proper grouping of lines into sections; it does this every time you give a command that uses the section structure in any way.

26.2 Compiling LISP Files

In the LISP Machine system, compilation and loading of LISP code is done with the functions **compile-file** and **load**. These operations do not really have anything to do with editing, but there are ZMACS commands to request them. The default file name for these commands is always the visited file name.

M-X Compile File

Compile the specified LISP source file into a QFASL file.

M-X Load File

Load the specified LISP source file or QFASL file.

M-X Compile And Load File

Compile the specified LISP source file into a QFASL file, unless the corresponding QFASL file exists already and is more recent than the source; then load the QFASL file.

You can also request compilation of files from the menu offered by **M-X Kill Or Save Buffers** (see chapter 19 [Buffers], page 99), or in Dired using the **A** command (see chapter 18 [Dired], page 91).

26.3 LISP Compilation and Evaluation

ZMACS can evaluate or compile LISP expressions directly from the ZMACS buffer, functions compiled in this way are immediately redefined in the LISP world.

C-Shift-E

Evaluate the current region, or the defun around or following point.

M-Shift-E

Similar, but print values as typeout, useful if you expect a lot of output.

C-M-Shift-E

Like **C-Shift-E**, but always reinitializes variables mentioned in **defvar** forms.

C-Shift-C

Compile the current region, or the defun around or following point.

C-M-Shift-C

Microcompile the current region, or the defun around or following point.

M-X Evaluate Buffer**M-X Compile Buffer**

Evaluate or compile the entire buffer.

M-Z Compile entire buffer, then exit ZMACS.

C-M-Z Evaluate entire buffer, then exit ZMACS.

M-X Compile Buffer Changed Sections**M-X Compile Changed Sections****M-X Tags Compile Changed Sections**

Compile all changed sections in one or more buffers.

M-X Evaluate Buffer Changed Sections**M-X Evaluate Changed Sections****M-X Tags Evaluate Changed Sections**

Evaluate all changed sections in one or more buffers.

Here "Evaluating" means that the expression is read with **read** and then passed to **eval**. "Compiling" in this case means that, if the LISP object returned by **read** is a call to **defun** or another function-defining special form or macro, a compiled function definition is installed rather than an interpreted one. "Compilation" of an expression other than a function definition is the same as evaluation.

ZMACS evaluation and compilation commands can apply to an arbitrary region, a single function definition, an entire buffer, or to all the changed definitions in one or more buffers. Normally you will load an entire program initially, either by compiling entire buffers or by loading files; then you will recompile individual LISP functions as you change them, or else change several and then ask to recompile all the changed ones.

All LISP reading and printing for these commands is done according to the syntactic attributes (**Readtable**, **Package** and **Base**) of the current buffer, unless the LISP code that is run binds or sets the relevant global variables (**zwei:*read-base***, etc.). See chapter 30 [Attributes], page 187. During evaluation, **zwei:*standard-output*** is set to print typeout, overlying the text display in the editing window, and **zwei:*standard-input*** is defined to echo its input as typeout.

The most general and basic evaluation/compilation commands are **C-Shift-E** (**Evaluate Region**) and **C-Shift-C** (**Compile Region**). If there is a region in effect, these commands act on the entire text of the region. Otherwise, the commands apply to the text of the defun (top-level LISP expression)

containing or following point. This implies that, after editing the definition of a LISP function, you can install the changed definition in the LISP world by typing simply **C-Shift-E** or **C-Shift-C**. (You might as well use **C-Shift-C** and compile it; interpreted functions are much slower, and have few advantages even for debugging.) See section 25.3 [Defuns], page 138.

C-Shift-E and **C-Shift-C** have one peculiar effect when there is no region: if the expression evaluated is or expands to a call to **defvar**, it does not evaluate normally. Normally, **defvar** sets the variable it defines only if the variable is void. When called from **C-Shift-E** and **C-Shift-C**, **defvar** sets the variable *unconditionally*. This is exactly what you want, if you alter the initial value in a **defvar** form and then evaluate that form.

Similar to **C-Shift-C** is the command **Control-Meta-Shift-C (Microcompile Region)**. Its only difference is that, after compiling a function definition, it goes on to microcompile the function. **C-Shift-C** microcompiles a function only if it contains a suitable declaration saying that it should always be microcompiled.

Two commands similar to **C-Shift-E** are **M-Shift-E (Evaluate Region Verbose)** and **C-M-Shift-E (Evaluate Region Hack)**. **M-Shift-E** differs by printing the values as typeout rather than in the echo area. This is useful if you expect the values to be large and really care about the values. **C-M-Shift-E** differs by always reinitializing variables for which **defvar** forms are found, even if used with a region.

The commands **M-X Compile Buffer** and **M-X Evaluate Buffer** are similar to the above commands, except that they operate always on the entire buffer. They do not treat **defvar** specially. **M-Z (Compile And Exit)** and **C-M-Z (Evaluate And Exit)** operate on the entire buffer and then continue to deselect the ZMACS frame and select the previously selected window. (This is what "exiting" means in the context of the LISP Machine, where switching programs means switching windows.)

Through the miracle of *sectionization*, ZMACS can remember which lines of the buffer belong to which LISP function; such a group of lines is called a *section*. See section 26.1 [Sectionization], page 153. ZMACS can additionally remember which sections have had their text altered since their last evaluation or compilation. Based on this information, it can evaluate or compile only the changed sections of a buffer. All the sections in a file are considered "unchanged" when the file is visited, and compiling or evaluating the full text of a section with any ZMACS command also marks it as "unchanged". A section becomes "changed" when any modification is made in its text.

The command **M-X Compile Buffer Changed Sections** compiles the text of each section of the current buffer that has changed since it was last compiled or evaluated and contains a definition form. Definition forms include function definitions, variable definitions, macro definitions, flavor

definitions, and so on. Non-definition forms are not executed, on the assumption that they may perform more drastic side-effects and possibly should not be executed twice. The command **M-X Evaluate Buffer Changed Sections** is similar, but evaluates function definitions rather than compiling them. A numeric argument to either command means to query about each changed section before operating on it.

Other versions of the **Changed Sections** commands apply to more than one buffer. **M-X Tags Compile Buffer Changed Sections** and **M-X Tags Evaluate Buffer Changed Sections** apply to all the buffers of LISP code in the current buffer group. See section 19.6 [Buffer Groups], page 104. **M-X Compile Changed Sections** and **M-X Evaluate Changed Sections** apply to all buffers of LISP code that exist in ZMACS.

26.4 Compiler Warnings

The LISP Machine compiler keeps a data base of compiler warnings, organized first by file name and then by function name within the file. Both file compilation and compilation from editor buffers use it. Each compilation discards from the data base any previous warnings that are obsolete. ZMACS can use the compiler warnings data base to display the pending warnings and find the functions that they apply to. See the section "Using Compiler Warnings" in the *LISP Machine Manual* for more information on the compiler warnings data base, including how to save warnings in a file and reload them for editing in a later session.

M-X Edit Warnings

Begin editing warnings of any or all of the files that have pending warnings. You are asked, for each file, whether to include it.

M-X Edit File Warnings

Begin editing warnings for one file, whose name you specify.

M-X Edit System Warnings

Begin editing warnings for all files in one system, whose name you specify.

C-Shift-W

Show the following warning and the function that it is about.

M-Shift-W

Show the previous warning and the function that it is about.

M-X Insert Warnings

Insert a list of all pending warnings of all files into the buffer at point.

M-X Insert File Warnings

Insert a list of all pending warnings of one specified file into the buffer at point.

While you are editing the pending warnings, the ZMACS frame is split into a small window at the top which displays one function's set of warnings, and a larger window below which displays the text of that function. Type **C-Shift-W (Edit Next Warning)** to move to the next function that has warnings; both windows move. Type **M-Shift-W (Edit Previous Warning)** to move to the previous function with warnings. The upper window's size is adjusted automatically to hold as many warnings as the function has, but there is a maximum size. If one function has too many warnings, they will not all fit in that maximum. Then you can use **C-M-V** to scroll the upper window to scroll through of the function's warnings.

To begin editing the pending warnings, you must specify which files' warnings you want to edit. Type **M-X Edit File Warnings** to edit the warnings of one file only; you specify the file with the minibuffer, and the file name defaults to the visited file's name. Type **M-X Edit System Warnings** to edit the warnings of all the source files in one system; you specify the system name with the minibuffer, and it defaults to the system that the visited file belongs to. For complete generality, type **M-X Edit Warnings**, which lists all the files for which warnings are recorded and asks, one by one, which ones you wish to process.

Once ZMACS knows which files you want to edit the warnings of, it creates a buffer containing a list of the currently pending warnings of those files, and then displays the first function listed and its warnings. You can then move to the next function with **C-Shift-W**. When you are done editing warnings, type **C-X 1** to remove the warnings window from display in the ZMACS frame.

You can correct the functions that got warnings, and recompile them, while editing the warnings, and this immediately removes them from the compiler warnings data base (or adds new warnings, if you make a mistake), but the buffer containing the warning list does not change until another **M-X Edit Warnings** or similar command is given. There is only one buffer of warnings in ZMACS, and each use of **M-X Edit Warnings** reinitializes it and starts at the beginning of it.

26.5 LISP Debugging Aids in ZMACS

M-X Find Unbalanced Parentheses

Scan entire buffer for mismatches.

M-X Where Is Symbol

List packages containing a symbol name (using LISP function **where-is**).

M-X List Matching Symbols

List LISP symbols matching specified pattern.

M-X Trace

Trace a LISP function.

M-X Disassemble

Print disassembly of a compiled LISP function.

M-X Macro Expand Expression**M-X Macro Expand Expression All**

Read the LISP expression following point, apply `macroexpand` or `macroexpand-all` to it, and print the results.

M-X List Callers**M-X List Object Users**

Print names of all LISP functions that refer to a symbol. Set up possibilities list of them also.

M-X Edit Callers**M-X Edit Object Users**

Similar, but don't print, and immediately visit the first possibility (caller).

M-X Multiple Edit Object Users**M-X Multiple Edit Callers****M-X Multiple List Object Users****M-X Multiple List Callers**

List or edit all functions referring to any of several symbols.

M-X Find Unbalanced Parentheses is the easiest way to check a file of LISP code for mismatches. It scans the entire current buffer and stops at the first place where a parenthesis appears to be missing. This is done automatically whenever a LISP mode buffer is saved if the variable `zwei:check-unbalanced-parentheses-when-saving*` is set non-`nil`. (It is `nil` by default.)

M-X Where Is Symbol is an interface to the LISP function `where-is`. It reads a string using the minibuffer and prints a list of LISP packages that contain symbols by that name. It also says which packages inherit each of the symbols.

M-X List Matching Symbols searches one or all packages for symbols that satisfy a predicate (a LISP function). The predicate is called repeatedly with one argument, a symbol, and the symbol is mentioned in the list if the predicate returns non-`nil`. With no numeric argument, **M-X List Matching Symbols** searches the current package. With `C-U` as argument, it searches all packages. With `C-U C-U` as argument, it reads a package name using the minibuffer and searches only that package. Example: with no numeric argument and `boundp` as the predicate, it prints a list of all symbols in the current package whose values are not void.

M-X Trace reads a function name using the minibuffer, and traces that function. The function name is defaulted and can be selected with the mouse, as in `Meta-..` **M-X Trace** pops up a menu with which you can select trace options, which go to construct a call to the LISP function `trace`.

You can see the **trace** call change in a window below the menu as you select options. Eventually, select **Abort** or **Do it**. **C-U M-X Trace** traces the function immediately, bypassing the menu step and using defaults for all trace options.

M-X Disassemble reads a function name using the minibuffer, and prints a disassembly of that function, which should be compiled LISP. The function name is defaulted and can be selected with the mouse, as in **Meta-..**

You can debug the expansion of LISP macros using **M-X Macro Expand Expression**. This reads the expression following point, passes it to **macroexpand**, to expand macro calls at the top level of that expression, and prints the result. The similar command **M-X Macro Expand Expression** uses **macroexpand-all**, which expands all macro calls found within the expression.

M-X List Callers scans all the functions in the LISP world and records which ones refer to a specified symbol, whose name is read with the minibuffer. Normally only functions that are definitions of symbols in the current package are scanned. (The current package is controlled by the **Package** attribute; see chapter 30 [Attribute], page 187) **C-U** as an argument says to search all packages. **C-U C-U** says to read the name of a package with the minibuffer and search only that one.


M-X List Object Users is like **M-X List Callers** only faster. It works from a precomputed data base. Initially the data base contains information on the files of the LISP Machine system. The function **si:analyze-all-files** adds the files you have loaded to the data base so that **M-X List Object Users** will include them. This command always covers all packages, it being fast enough to do so.

Both **M-X List Object Users** and **M-X List Callers** print mouse-sensitive function names, so that you can visit the source code for the definition of any of them by clicking **(L)** on it. In addition, they set up possibilities lists containing all the names found, enabling you to visit the definitions one by one using **C-Shift-P**. See section 26.7 [Possibilities Lists], page 164.

M-X Edit Object Users and **M-X Edit Callers**, which move immediately to the definition of the first caller, also exist.

Since it is as quick to all find the users of several symbols as it is to find all the users of one, *multiple* commands are provided to find the users of several symbols at once. Their names are **M-X Multiple List Callers**, etc. These commands use the minibuffer to read the symbols; enter each symbol as a separate minibuffer argument. Enter an empty argument to mark the end of the arguments. This says that the command should proceed.

26.6 Exploring the Flavor Hierarchy

This section describes the ZMACS commands for printing information about flavor definitions in effect in the LISP world. All the flavor names and method function specs that they print are mouse-sensitive; click  on one to visit the source code for its definition.

M-X List Combined Methods

List all the methods that play a part in handling a specified operation on a specified flavor. This includes inherited methods.

M-X List Methods

List all methods defined, on whatever flavor, for a specified operation.

M-X List Flavor Components

List all the component flavors of a specified flavor.

M-X List Flavor Dependents

List all the flavors that depend on a specified flavor.

M-X List Flavor Direct Dependents

List all the flavors that depend directly on a specified flavor.

M-X List Flavor Methods

List all methods defined on a specified flavor, not including inherited ones.

M-X Edit Combined Methods

M-X Edit Methods

M-X Edit Flavor Components

M-X Edit Flavor Dependents

M-X Edit Flavor Direct Dependents

M-X Edit Flavor Methods

Similar, but immediately visit the first flavor or method in the list.

M-X Describe Flavor

Print everything there is to know about a specified flavor.

Each of these commands except **M-X Describe Flavor** sets up a possibilities list containing all of the flavors or methods in the list. This allows you to visit each of the flavors, or each of the methods, one by one with the **C-Shift-P** command. See section 26.7 [Possibilities Lists], page 164. With the **Edit** commands, the possibilities list is the only way you can see the names of, or visit the source code for, flavors or methods aside from the first one.

26.7 Possibilities Lists

A *possibilities list* is a list of functions, flavors, or whatever, which ZMACS sets up so that you can look at the source code for each one, as you are ready for them. For example, the command functions **List Flavor Components** and **Edit Flavor Components** create a possibilities list containing all the component flavors of a specified flavor. After either of them, you can use **C-Shift-P** (**Go To Next Possibility**) repeatedly to visit the source code for the next flavor in the list.

C-Shift-P

Visit text specified by the next possibility in the current possibilities list.

List Flavor Components, and other such commands whose names start with **List**, prints a mouse-sensitive list of all the objects it has found. **Edit Flavor Components**, and other such commands whose names start with **Edit**, immediately performs **C-Shift-P** automatically to visit the first flavor; the first time you type **C-Shift-P** explicitly, you visit the second one.

The current possibilities list, and all previous ones you have made, live in the ZMACS buffer named **Possibilities**. Each possibilities list occupies a separate page, with the most recently created ones coming first in the buffer. Each possibility is described by a line. Your current possibilities list and "place" in it are indicated by point: point at the beginning of a line means that line is next; point at the end of the line, that that line has been done and the following possibility-line is next. **C-Shift-P** takes successive possibilities by moving point to the end of the line that it uses.

You can use the possibilities out of order, or go back to an possibilities list, by selecting the buffer **Possibilities** and moving point to the beginning of the next possibility you want to use.

26.8 The 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 both the LISP Machine system itself and applications systems that are large enough to be worth loading up and saving on a disk partition.

Since it is very inconvenient to load a large system each time it is to be used, often LISP bands containing the system are saved. This means that a bug that is fixed will remain in the saved bands even after the files have been recompiled. This problem can be solved by putting fixes into *patch files*, files of incremental changes that are loaded into previously saved LISP bands each time they are booted.

A patch file contains the **Patch File** attribute in its attribute list. See chapter 30 [Attributes], page 187. The effect of this attribute is that when the file is loaded there will be no warning about redefining functions or variables previously defined in other files. After all, the purpose of a patch file is to do just that. In fact, each piece of text in a patch file says which source file the text was patched from; when the patch file is loaded, the definitions are recorded as coming from that source file rather than from the patch file itself.

Each *patchable system* has a *major version number* which is incremented from time to time when it is recompiled. Each major version has a series of patch files, given sequential *minor version numbers*, which contain the changes that have been patched since that major version was created. A given LISP band containing a system loads only the patches for the major version that it contains; patches for earlier major versions are not needed, since the changes they contain were presumably included in the original files for the current major version. See the section "The Patch Facility" in the *LISP Machine Manual* for more information on patchable systems and loading patches.

Patches are made with special ZMACS commands, usually by copying text out of the source file after you change it. Each patch must be associated with a specific patchable system, and had better contain changes only to the source files of that system. You must specify the patchable system name when you start the patch file.

26.8.1 Making Patches

During a typical maintenance session on a system you will make several edits to its source files. The ZMACS patch commands can be used 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 modified function definitions, new functions, modified **defvar** and **defconst** forms, or arbitrary forms to be evaluated, even including **load** of new files.

M-X Start Patch

Begin editing a patch file.

M-X Add Patch

Copy region or current defun to patch file being edited.

M-X Add Patch Changed Sections

M-X Add Patch Buffer Changed Sections

M-X Tags Add Patch Changed Sections

Copy changed sections of one or more buffers into patch file being edited.

M-X Finish Patch

Install the patch file being edited for users to load and save.

M-X Finish Patch Unreleased

Install the patch file being edited for users to test but not save.

M-X Release Patch

Permit users to save a patch file previously finished but not released.

M-X Resume Patch

Start editing an existing patch file.

M-X Cancel Patch

Eliminate a patch file from the patch data base.

The first step in making a patch is to *start* it. At this stage you must specify which patchable system you are making a patch for. Then you *add* one or more pieces of code from other source files to the patch. Finally you *finish* the patch. This is when you fill in the description of what the patch does; this description is what **load-patches** prints when it offers to load the patch. If you have any doubts about whether the patch will load and work properly, you finish it *unreleased*; then you can load it to test it, but no bands can be saved containing the patch until you explicitly release it later.

It is important that any change you patch should go in a patch for the patchable system to which the changed source file belongs. This makes sure that nobody loads the change into a LISP world that does not contain the file you were changing—something that might cause trouble. Also, it ensures that you never patch changes to the same piece of code in two different patchable systems' patches. This would lead to confusion because there is no constraint on the order in which patches to two different systems are loaded.

Starting a patch can be done with **M-X Start Patch**. It reads the name of the system to patch with the minibuffer. **M-X Add Patch** also starts a patch if you are not already editing one, so an explicit **M-X Start Patch** is needed only infrequently.

M-X Add Patch adds the region (if there is one) or the current defun to the patch file currently being constructed. If you change a function, you should recompile it, test it, then once it works use **M-X Add Patch** to put it in the patch file. If no patch is being constructed, one is started for you; you must type in the name of the system to patch.

A convenient way to add all your changes to a patch file is to use **M-X Add Patch Changed Sections** or **M-X Add Patch Buffer Changed Sections**. These commands ask you, for each changed function (or each changed function in the current buffer), whether to add it to the patch being constructed. If you use these commands more than once, a function that has been added to the patch and has not been changed since is considered "unchanged".

The patch file is constructed in an ordinary ZMACS buffer. If you mistakenly **M-X Add Patch** something that doesn't work, you can select the buffer containing the patch file and delete it. Then later you can **M-X Add Patch** the corrected version.

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. If two people are patching a system at the same time, they do not both get the same minor version number.

After testing and patching all of your changes, use **M-X Finish Patch** to install the patch file so that other users can load it. This compiles the patch file if you have not done so already (patches are always compiled). It also asks you for a comment describing the reason for the patch; **load-patches** and **print-system-modifications** print these comments. If the patch is complex or it has a good chance of causing new problems, you should not use **M-X Finish Patch**; instead, you should make an unreleased patch.

A finished patch can be *released* or *unreleased*. If a patch is unreleased, it can be loaded in the usual manner if the user says 'yes' to a special query, but once it has been loaded the user will be strongly discouraged from saving a band. Therefore, you still have a chance to edit the patch file and recompile it if there is something wrong with it. You can be sure that the old broken patch will not remain permanently in saved bands.

To finish a patch without releasing it, use the command **M-X Finish Patch Unreleased**. Then the patch can be tested by loading it. After a sufficient period for testing, you can release the patch with **M-X Release Patch**. If you discover a bug in the patch after this point, it is not sufficient to correct it in this patch file; you must put the fix in a new patch to correct any bands already saved with the broken version of this patch.

It is a good principle not to add any new features or fix any additional bugs in a patch once that patch is finished; change it only to correct problems with that patch. New fixes to other bugs should go in new patches.

You can only be constructing one patch at any time. **M-X Add Patch** automatically adds to the patch you are constructing. But if you use the command **M-X Start Patch** while constructing a patch, you are given the option of starting a new patch. The old patch ceases to be the one you are constructing but the patch file remains in its editor buffer. Later, or in another session, you can go back to constructing the first patch with the command **M-X Resume Patch**. This command asks for both a patchable system name and the patch version to resume constructing. You can simply save the editor buffer of a patch file and resume constructing that patch in a later session. You can even resume constructing a finished patch; though it rarely makes sense to do this unless the

patch is unreleased.

If you start to make a patch and change your mind, use the command **M-X Cancel Patch**. This deletes the record that says that this patch is being worked on. It also tells the editor that you are no longer constructing any patch. You can undo a finished (but unreleased) patch by using **M-X Resume Patch** and then **M-X Cancel Patch**. If a patch is released, you cannot remove it from saved bands, so it is not reasonable to cancel it at that stage.

26.8.2 Private Patches

A private patch is a file of changes that is not installed. It is loaded only by explicit request (using the function `load`). A private patch is not associated with any particular patchable system, and has no version number.

M-X Start Private Patch

Begin editing a patch file that will not be installed for all users.

To make a private patch, use the editor command **M-X Start Private Patch**. Instead of a patchable system name, you must specify a file name to use for the patch file; since the patch is not to be installed, there is no standard naming convention for it to follow. Add text to the patch using **M-X Add Patch** and finish it using **M-X Finish Patch**. There is no concept of release for private patches so there is no point in using **M-X Finish Patch Unreleased**. There is also no data base recording all private patches, so **M-X Start Private Patch** will resume an existing patch, or even a finished patch. In fact, finishing a private patch is merely a way to write a comment into it and compile it.

Once the private patch file is made, you can load it like any other file.



27. Ztop Mode

Ztop Mode enables ZMACS to serve the purpose of a LISP Listener, with editing capability on both previous input and previous output. Ztop uses a ZMACS buffer as a typescript that records the LISP expressions that you evaluate, the input that they read while executing, the output that they print, and the values they return. Text can be copied into the Ztop buffer from other ZMACS buffers used as input there; it can also be copied from input or output earlier in the Ztop buffer. Nothing is ever deleted from the Ztop buffer unless you do so explicitly; therefore, the Ztop buffer comes to contain a typescript of the entire session.

M-X Ztop Mode

Put current buffer in Ztop mode.

M-X Select Last Ztop Buffer

Select a buffer that is in Ztop mode, creating a new buffer if no such buffer exists.

M-X Require Activation Mode

Tell Ztop not to activate input on self-inserting characters.

END In Ztop mode, activate input.

C-M-Y In Ztop mode, copy previous input at end of buffer for resubmission.

ABORT In Ztop mode, abort the program that is executing, if it is reading input.

M-ABORT In Ztop mode, abort the program that is executing all the way to top level, if it is reading input.

To begin using Ztop, make a buffer with **C-X B bufname RETURN RETURN** and put it in Ztop mode with **M-X Ztop Mode**. Alternatively, use **M-X Select Last Ztop Buffer**, which selects an existing Ztop buffer or creates a new one. Now you are ready to enter LISP expressions to be evaluated. Note that two **RETURN** characters are needed when creating a buffer. It is often useful to split the ZMACS frame into two windows, select the Ztop buffer in one of them, and use the other for editing source files.

To evaluate an expression with Ztop, move to the end of the Ztop buffer and insert the text of the expression. Ztop can tell automatically when you have typed a complete expression, just as a LISP Listener can; as soon as you have done so, the expression is evaluated, and its values are printed as text into the Ztop buffer. After the values are printed, Ztop is ready for you to insert another expression.

For example, if you go to the end of the Ztop buffer and type **(cons 'a 'b)**, the expression will be evaluated immediately and the output **RETURN (a . b) RETURN** will be inserted. This leaves point at the end of the buffer, so you can immediately type another expression.

The input in Ztop mode is not restricted to LISP expressions. If your LISP expression calls a program that reads input in some other language, that input too is read through the Ztop buffer. This is just like the way LISP Listener windows work. Specifically, ***terminal-io*** in the program is an *editor stream* through which input is read out of the Ztop buffer and output is inserted into the buffer. Printing values of expressions is a special case of output to ***terminal-io***; it is the effect of the LISP listen loop calling **prin1**. Aside from editing, any sequence of text that works in the LISP Listener works the same way in Ztop, except for programs that use graphics operations such as **:draw-line** on ***terminal-io***. Ztop does not support these.

Both input and output in the Ztop buffer use a buffer pointer called the *I/O pointer*. Input advances the I/O pointer over characters already in the buffer. If the program needs to read input and the I/O pointer is at the end of the buffer, the program waits until you insert more text after the I/O pointer. Output is inserted into the buffer at the I/O pointer, advancing the pointer beyond it so that the output will not be mistakenly read as the next input. The result is that input and output ultimately are in logical order in the Ztop buffer: the output from an expression appears after that expression, and before the next expression, even if the following expression was inserted before the output was printed.

As long as point is at the end of the Ztop buffer, any text you insert is passed to the program (LISP listen loop, or other) after each self-inserting character you type. Therefore, the I/O pointer keeps up with the end of the buffer. However, until a complete expression or other unit of input has been typed, you can still rub out the input.

If a program that reads single-character commands, such as the debugger, is invoked using Ztop, the commands of this program override those of ZMACS. Thus, **Control-E** would be taken as the debugger command to edit the definition of the function of the current frame, rather than as the ZMACS command to move to the end of the line. But this is true only if point is at the end of the buffer, and only as long as the single-character command is valid. For example, if in the debugger you type at least one character of a LISP expression, the debugger ceases to be interested in single character commands. As a result, these characters regain their normal ZMACS meanings.

If you move point away from the end of the Ztop buffer, or if you yank or insert in any way except by typing text, execution becomes *dormant*. The I/O pointer is moved back to the beginning of the current expression, and becomes visible as a blinking I-shape. You can edit the text after the I-blinker; when the text is finally used as input, it will be used in the changed form. However, this cannot happen until you do something to *activate* input again. You can also edit the text before the I-blinker, but that text will not be used as input unless it is copied to a place after the I-blinker.

Normally, the way to activate is to move point to the end of the buffer and type a self-inserting character. Type **(SPACE)** or **(RETURN)**, if you have no reason to insert meaningful text. Activation causes the I-blinker to disappear. If the input now forms a complete expression, it will be executed immediately; otherwise, you must insert more input, and you have further opportunity to edit the input.

Another way to activate Ztop when it is dormant is to type **(END)** (**Finish Ztop Evaluation**). This moves point to the end of the buffer and activates. If there is an active region when **(END)** is typed, the text of the region is copied to the end of the buffer before activating input. This implies that that text will be read as input.

Require Activation mode is a minor mode that can be used along with Ztop mode. It causes **(END)** to be the only way to activate Ztop input if it becomes dormant. In Require Activation mode, you can still enter input without using **(END)** as long as point remains at the end of the buffer and nothing else is done to make Ztop dormant; but once Ztop is dormant, it stays dormant until **(END)** is typed. The command **M-X Require Activation Mode** toggles Require Activation mode; with numeric argument, it turns the mode on if the argument is positive, off otherwise.

C-M-Y (Ztop Yank Input History) can be used in Ztop mode to yank previous units of input. A "unit of input" here means the largest piece of text within which you can still rub out. Ztop remembers each unit of input in a history, as well as in the Ztop buffer itself, and **C-M-Y** yanks from this history. Thus, just **C-M-Y** yanks the most recent previous input unit, **C-M-Y** with a numeric argument *n* yanks the *n*'th from last input unit, and **M-Y** can be used to replace one yanked input unit with a previous one until you find the one you wanted. See section 11.2 [Yanking], page 47.

(ABORT) and **M-(ABORT)** are defined in Ztop mode as commands that abort the program execution when they are read. The result is that these characters have effectively the same behavior as they do when typed at a LISP Listener.

Setting the Ztop buffer's current package in ZMACS using **Set Package** automatically informs the program that is reading from Ztop to use the new package. Conversely, setting the current package by executing a program (such as, calling **pkg-goto**) informs ZMACS to change the Ztop buffer's current package. See chapter 30 [Attributes], page 187.

Ztop works by running the LISP listen loop in a specially created stack group, and giving it a ***terminal-io*** stream that uses ZMACS as the input editor.



28. Word Abbrevs

A *word abbrev* is a word that changes (*expands*), if you insert it, into some different text. Abbrevs are defined by the user to expand in specific ways. For example, you might define 'foo' as an abbrev expanding to 'find outer otter'. With this abbrev defined, you would be able to get 'find outer otter' into the buffer by typing `f o o (SPACE)`.

Abbrevs expand only when Word Abbrev mode (a minor mode) is enabled. Disabling Word Abbrev mode does not cause abbrev definitions to be forgotten, but they do not expand until Word Abbrev mode is enabled again. The command `M-X Word Abbrev Mode` toggles Word Abbrev mode; with a numeric argument, it turns Word Abbrev mode on if the argument is positive, off otherwise. See section 31.1 [Minor Modes], page 191.

Abbrev definitions can be *mode-specific*—active only in one major mode. Abbrevs can also have *global* definitions that are active in all major modes. The same abbrev can have a global definition and various mode-specific definitions for different major modes. A mode specific definition for the current major mode overrides a global definition.

Word abbrevs can be defined interactively during the editing session. Lists of abbrev definitions can also be saved in files and reloaded in later sessions. Some users keep extensive lists of abbrevs that they load in every session.

28.1 Defining Abbrevs

`C-X +` Define an abbrev to expand into some text before point.

`C-X C-A` Define an abbrev available only in the current major mode.

`M-X Make Word Abbrev`

General way to define an abbrev.

`M-X Kill Global Word Abbrev`

Remove global definition of an abbrev.

`M-X Kill Mode Word Abbrev`

Remove mode-specific definition of an abbrev.

`M-X Kill All Word Abbrevs`

After this command, there are no abbrev definitions in effect.

The usual way to define an abbrev is to enter the text you want the abbrev to expand to, position point after it, and type `C-X + (Add Global Word Abbrev)`. This reads the abbrev itself

using the minibuffer, and then defines it as an abbrev for one or more words before point. Use a numeric argument to say how many words before point should be taken as the expansion. For example, to define the abbrev 'foo' as mentioned above, insert the text 'find outer otter' and then type **C-U 3 C-X + f o o** **RETURN**.

If there is a region, that is used as the expansion for the abbrev that is defined.

The command **C-X C-A (Add Mode Word Abbrev)** is similar, but defines a mode-specific abbrev. Mode specific abbrevs are active only in a particular major mode. **C-X C-A** defines an abbrev for the major mode in effect at the time **C-X C-A** is typed.

M-X Make Word Abbrev is a general way of defining abbrevs. A numeric argument says to make a global abbrev; otherwise, a mode-specific abbrev for the current major mode is defined. The expansion and then the abbrev are read separately using the minibuffer.

To change the definition of an abbrev, just add the new definition. The old definition is replaced automatically. To remove an abbrev definition and leave none, so that the word is no longer an abbrev, use **M-X Kill Mode Word Abbrev** or **M-X Kill Global Word Abbrev**. You must choose the command to specify whether to kill a global definition or a mode-specific definition for the current mode, since those two definitions are independent for one abbrev.

M-X Kill All Word Abbrevs removes all the abbrev definitions there are.

28.2 Expanding Abbrevs

An abbrev expands whenever it is present in the buffer just before point and a self-inserting punctuation character (**SPACE**, **COMMA**, etc.) is typed. Most often the way an abbrev is used is to insert the abbrev followed by punctuation.

Abbrev expansion preserves case; thus, 'foo' expands into 'find outer otter'; 'Foo' into 'Find outer otter', and 'FOO' into 'FIND OUTER OTTER'.

These two commands are used to control abbrev expansion:

- M-'** Separate a prefix from a following abbrev to be expanded.
- C-X U** Undo last abbrev expansion.

You may wish to expand an abbrev with a prefix attached; for example, if 'cnst' expands into 'construction', you might want to use it to enter 'reconstruction'. It does not work to type `recnst`, because that is not necessarily a defined abbrev. What does work is to use the command `M-` (Word Abbrev Prefix Mark) in between the prefix 're' and the abbrev 'cnst'. First, insert 're'. Then type `M-`; this inserts a minus sign in the buffer to indicate that it has done its work. Then insert the abbrev 'cnst'; the buffer now contains 're-cnst'. Now insert a punctuation character to expand the abbrev 'cnst' into 'construction'. The minus sign is deleted at this point, because `M-` left word for this to be done. The resulting text is the desired 'reconstruction'.

If you actually want the text of the abbrev in the buffer, rather than its expansion, you can accomplish this by inserting the following punctuation with `C-Q`. Thus, `foo C-Q` - leaves 'foo-' in the buffer.

If you expand an abbrev by mistake, you can undo the expansion (replace the expansion by the original abbrev text) with `C-X U` (Unexpand Last Word). `C-Shift-U` can also be used to undo the expansion; but first it will undo the insertion of the following punctuation character!

28.3 Examining and Altering Abbrevs

`M-X List Word Abbrevs`

Print a list of all abbrev definitions.

`M-X List Some Word Abbrevs`

Print all abbrev definitions where the abbrev or expansion contains a given string.

`M-X Edit Word Abbrevs`

Edit a list of abbrevs; you can add, alter or remove definitions.

The output from `M-X List Word Abbrevs` looks like this:

```
PRN:                0      "parentheses"
FOO:      (LISP)     5      "Find outer otter"
```

The word at the beginning is the abbrev; the word in parentheses is the name of the major mode it is in effect for, or absent for a global definition. The number that appears is the number of times the abbrev has been expanded. ZMACS keeps track of this to help you see which abbrevs you actually use, in case you decide to eliminate those that you don't use often. The string at the end of the line is the expansion.

M-X List Some Word Abbrevs is a sort of apropos for abbrevs. It prints a list of definitions like **M-X List Word Abbrevs**, but includes only abbrevs that contain a specified string, or whose expansions contain that string.

M-X Edit Word Abbrevs allows you to add, change or kill abbrev definitions by editing a list of them in a recursive edit. The list has the same format described above. After making changes as you like, type **END** to exit the recursive edit and put the changes into effect, or type **ABORT** to exit the recursive edit and ignore the editing you did in it (no abbrev definitions are changed). See section 29.1 [Recursive Edit], page 179.

28.4 Saving Abbrevs

M-X Write Word Abbrev File

Write a file describing all defined word abbrevs.

M-X Read Word Abbrev File

Read such a file and define abbrevs as specified there.

M-X Define Word Abbrevs

Define abbrevs from buffer.

M-X Insert Word Abbrevs

Insert all abbrevs and their expansions into the buffer.

M-X Write Word Abbrev File reads a file name using the minibuffer and writes a description of all current word abbrev definitions into that file. The text stored in the file looks like the output of **M-X List Word Abbrevs**, sans the header line.

M-X Read Word Abbrev File reads a file name using the minibuffer and reads the file, defining word abbrevs according to the contents of the file.

These commands are used to save word abbrev definitions for use in a later session.

The commands **M-X Insert Word Abbrevs** and **M-X Define Word Abbrevs** are similar to the previous commands but work on text in a ZMACS buffer. **M-X Insert Word Abbrevs** inserts text into the current buffer before point, describing all current word abbrev definitions; **M-X Define Word Abbrevs** parses the entire current buffer and defines abbrevs accordingly.

29. Miscellaneous Commands

This chapter contains several brief topics that do not fit anywhere else.

29.1 Recursive Edits

A *recursive edit* is a situation in which you are using ZMACS commands to perform arbitrary editing while in the middle of another ZMACS command. For example, when you type **C-R** inside of a **Query Replace**, you enter a recursive edit and can change the current buffer.

Exiting the recursive edit means returning to the unfinished command, which continues execution. For example, exiting the recursive edit requested by **C-R** in **Query Replace** causes query replacing to resume. Exiting is done with **(END)**.

You can also *abort* the recursive edit. This is like exiting, but the unfinished command is immediately aborted, and you end up at top level. See section 32.1 [Quitting], page 205.

The text being edited inside the recursive edit need not be the same text that you were editing at top level. It depends on what the recursive edit is for. **Edit Tab Stops** is one example of a command function that enters a recursive edit on text (the tab stop buffer) which is different from what you were editing at top level.

29.2 Sending Mail

You cannot currently read mail with ZMACS, but you can send mail.

C-X M Begin composing a message to send.

The command **C-X M (Mail)** creates a buffer named **Mail-n** and initializes it with the skeleton of an outgoing message. This is the beginning of the process of using ZMACS to send a message. The rest of the process is to fill in the addressees, subject and text of the message, and then type **(END)** to send it.

The line in the buffer that says

--Text follows this line--

is a special delimiter. Whatever follows it is the text of the message; the headers precede it. The delimiter line itself does not appear in the message.

Header fields you can use include 'To', 'From', 'Subject' and 'CC'. They look like this:

```
To: rms@mc
CC: mly@mc, rg@oz
Subject: The ZMACS Manual
```

'Subject:' can be abbreviated 'S:'.

Because the mail composition buffer is an ordinary ZMACS buffer, you can switch to other buffers while in the middle of composing mail, and switch back later (or never). You can be composing several messages at once, because each use of **C-X M** makes a new *Mail-*n** buffer for a new *n*. Old *Mail-*n** buffers are reused only if their messages have been sent.

ZMail templates can be used when sending mail in ZMACS. Each template defined is an extended command and can be invoked with **M-X**. Also, you can specify a template to be invoked automatically. The value of the variable `zwei:*default-ZMACS-mail-template*`, if non-nil, is taken to be a template to invoke as soon as a mail buffer is initialized for **C-X M**. The value of `zwei:*default-ZMACS-bug-template*` is used likewise in buffers made by **M-X Bug**. Refer to the *ZMail Manual* for more information on templates.

29.3 Hardcopy Output

M-Shift-P (Quick Print Buffer)

Print hardcopy of the current buffer.

M-X Print Buffer

Print hardcopy of a specified buffer.

M-X Print All Buffers

Ask about each buffer, and then print hardcopy of the chosen buffers.

M-X Print Region

Print hardcopy of the contents of the region.

M-X Print File

Print hardcopy of the specified file.

The ZMACS hardcopy commands, except for **M-X Print File**, all use **hardcopy-stream** to print the specified text out of the buffer or buffers. They differ only in deciding what text to print. **M-X Print File** uses **hardcopy-file** instead. All the commands default all arguments except the file name or stream, and the name to put on the header page.

When a multi-font buffer is printed, the hardcopy device will, if it knows how, choose its fonts based on the fonts used in the buffer. Printing a file does the same thing based on the file's **Fonts** and **Vsp** attributes, if it has them. See chapter 30 [Attributes], page 187.

29.4 Sorting

There are several commands to divide the region into *sort records* and rearrange them in alphabetical order. They differ in how the sort records are defined.

M-X Sort Lines

Sort the region line by line.

M-X Sort Paragraphs

Sort the region paragraph by paragraph.

M-X Sort Pages

Sort the region page by page.

M-X Sort Via ^{Keyboard}~~Kbd~~-Macros

Use user-specified keyboard macros to divide the region into sort records.

M-X Sort Lines takes each line in the region as a sort record. The lines are rearranged into alphabetical order, but not otherwise changed. **M-X Sort Paragraphs** similarly takes each paragraph in the region as a sort record. Paragraph-separating lines, such as blank lines, accompany the following paragraph to its final position, but are ignored when comparing the text of two paragraphs. See section 24.4 [Paragraphs], page 128.

M-X Sort Pages is like the previous two commands, but has some peculiarities that go with the need for an explicit page delimiter character to start a new page. Every page except the last one must end with a page delimiter character, but the last page ends at the end of the region with or without a page delimiter. If the last page does not end with a delimiter and moves to a different position as a result of sorting, a page delimiter character is inserted to terminate it, and the page delimiter is deleted from the end of whichever page winds up last. See section 24.5 [Pages], page 129.

M-X Sort Via ^{Keyboard}~~Kbd~~ Macros is a very general sort command that could be used to sort by lines, paragraphs, pages or other units of text. To use it, consider the region as divided up into sort records, each of which contains a *sort key*. Each region is moved as a unit to its final position, but only its sort key is used to determine that position. Use of **M-X Sort Via ^{Keyboard}~~Kbd~~ Macros** requires you to tell it how to find the sort key in a sort record, and how to find the end of the sort record. You do this with three keyboard macros:

A macro that, given point at the beginning of a sort record, positions point at the beginning of the record's key.

A macro that, given point at the beginning of the sort key, positions point at the end of the key.

A macro that, given point at the end of the sort key, positions point at the end of the containing record.

After you have entered all three macros, the region is divided up into records, which are then rearranged. See section 31.3 [Keyboard Macros], page 195. End each macro with **C-X)** as usual.

An example of using this command is

M-X Sort Via ^{Keyboard}~~Kbd~~ Macros (**RETURN**) C-X) C-N C-X) C-X)

which has the same effect as **M-X Sort Lines** (assuming that both ends of the region are at the beginnings of lines). Each sort records's key begins at the beginning of the record, and the key and the record end at the start of the next line.

29.4.1 Evaluating Expressions Interactively

M-ALTMODE

Read an expression using the minibuffer, then evaluate it.

M-X Evaluate Into Buffer

Similar, but insert the text of the value into the buffer before point.

M-X Evaluate And Replace Into Buffer

Evaluate and delete the expression after point, inserting value instead.

BREAK

Enter a read-eval-print loop, using timeout for output and for echoing input.

M-ALTMODE (**Evaluate Mini Buffer**) reads a LISP expression using the minibuffer, and evaluates it, printing the value in the echo area. Since it is often useful to make the expression more than

one line long, **RETURN** is defined to insert newlines as it does at top level. You must terminate the expression with **END**.

M-X Evaluate Into Buffer works much like **M-ALTMODE**, but "prints" its value(s) into the current buffer, inserting the text before point. A newline is inserted before each value. A numeric argument to this command causes output printed by the expression on **zwei:*standard-output*** to be inserted in the buffer as well. Such output precedes the values in the buffer.

M-X Evaluate And Replace Into Buffer reads the LISP expression in the buffer following point, and evaluates it. On successful evaluation, it deletes the text of the expression that was read, and inserts the value in its place.

BREAK (running the command function **Break**) invokes a break loop, a read-eval-print loop like that in the LISP Listener window. It runs as a subroutine of ZMACS, using the ZMACS typeout window for both input and output. Type **RESUME** or **ABORT** to get back to ZMACS.

All LISP reading and printing for these commands is done according to the syntactic attributes (**Readable**, **Package** and **Base**) of the current buffer, unless the LISP code that is run binds or sets the relevant global variables (**zwei:*read-base***, etc.). See chapter 30 [Attributes], page 187. During evaluation, **zwei:*standard-output*** is set to print typeout, overlying the text display in the editing window, and **zwei:*standard-input*** is defined to echo its input as typeout.

29.5 Editing Assembly-Language Programs

MIDAS mode is designed for editing programs written in the MIDAS assembler. It is suitable for many other assemblers as well. To select it, use **M-X Midas Mode**.

In MIDAS mode, comments start with '**;**', and '**<**' and '**>**' have the syntax of parentheses, as do square brackets and curly braces. In addition, there are five special commands that understand the syntax of instructions and labels. These commands are:

- C-M-N** Go to Next label.
- C-M-P** Go to Previous label.
- C-M-A** Go to Accumulator field of instruction.
- C-M-E** Go to Effective Address field.
- C-M-D** Kill next word and its Delimiting character.

Two other commands that behave slightly differently in MIDAS mode are

- M-[** Move up to previous blank line.
- M-]** Move down to next blank line.

Any line that is not indented and is not just a comment is taken to contain a *label*. The label is everything up to the first whitespace (or the end of the line). **C-M-N** (**Go to Next Label**) and **C-M-P** (**Go to Previous Label**) both position the cursor right at the end of a label; **C-M-N** moves forward or down and **C-M-P** moves backward or up. At the beginning of a line containing a label, **C-M-N** moves past it. Past the label on the same line, **C-M-P** moves back to the end of it. If you kill a couple of indented lines and want to insert them right after a label, these commands put you at just the right place.

C-M-A (**Go to AC Field**) and **C-M-E** (**Go to Address Field**) move to the beginning of the accumulator (AC) or effective address fields of a PDP-10 instruction. They always stay on the same line, moving either forward or backward as appropriate. If the instruction contains no AC field, **C-M-A** positions to the start of the address field. If the instruction is just an opcode with no AC field or address field, a space is inserted after the opcode and the cursor left after the space. In PDP-11 programs, **C-M-A** moves to the first operand and **C-M-E** moves to the second operand.

Once you've gone to the beginning of the AC field you can often use **C-M-D** (**Kill Terminated Word**) to kill the AC name and the comma that terminates it. You can also use it at the beginning of a line, to kill a label and its colon, or after a line's indentation to kill the opcode and the following space. This is very convenient for moving a label from one line to another. In general, **C-M-D** is equivalent to **M-D C-1 C-D**, since the numeric argument causes **C-D** to save the character it kills on the kill history.

The **M-[** and **M-]** commands are not, strictly speaking, redefined by MIDAS mode. They go up or down to a paragraph boundary, as usual. However, in MIDAS mode the criterion for a paragraph boundary is changed by setting the variable `zwei:*paragraph-delimiter-list*` to `nil` (see section 24.4 [Paragraphs], page 128) so that only blank lines (and page boundaries) delimit paragraphs. So, **M-[** moves up to the previous blank line and **M-]** moves to the next one.

29.6 Major Modes for Other Languages

MACSYMA mode redefines the syntax of words and s-expressions in an attempt to make it easier to move over MACSYMA syntactic units. **TAB** is defined to run the command function **Indent Nested**. Also, the syntax of MACSYMA comments is understood.

PL1 mode is for editing PL1 code, and causes **(TAB)** to indent an amount based on the previous statement type. The body of the implementation of PL1 mode is in the library PL1, which is loaded automatically when necessary.

C mode is for editing C code. It assumes that indentation levels in the code are 8 columns apart, so **(TAB)** just inserts a tab character. The syntax of C comments is understood.

29.7 Dissociated Press

M-X Dissociated Press is a command for scrambling a file of text either word by word or character by character. Starting from a buffer of straight English, it produces extremely amusing output. The input comes from a ZMACS buffer; you must specify the input buffer name using the minibuffer. Dissociated Press prints its output as "typeout"; it does not change the contents of the buffer. However, you can route the output into a buffer, using **M-X Execute Command Into Buffer**, if you wish to record it permanently. See section 1.2 [Typeout], page 5.

Dissociated Press operates by jumping at random from one point in the buffer to another. In order to produce plausible output rather than gibberish, it insists on a certain amount of overlap between the end of one run of consecutive words or characters and the start of the next. That is, if it has just printed out 'president' and then decides to jump to a different point in the file, it might spot the 'ent' in 'pentagon' and continue from there, producing 'presidentagon'. Long sample texts produce the best results.

A positive argument to **M-X Dissociated Press** tells it to operate character by character, and specifies the number of overlap characters. A negative argument tells it to operate word by word and specifies the number of overlap words. In this mode, whole words are treated as the elements to be permuted, rather than characters. No argument is equivalent to an argument of two. For your aginformation, the output is only printed on the screen. The file you start with is not changed.

Dissociated Press produces nearly the same results as a Markov chain based on a frequency table constructed from the sample text. It is, however, an independent, ignoriginal invention. Dissociated Press techniquitously copies several consecutive characters from the sample between random choices, whereas a Markov chain would choose randomly for each word or character. This makes for more plausible sounding results.

It is a mustatement that too much use of Dissociated Press can be a developediment to your real work. Sometimes to the point of outragedy. And keep dissociwords out of your documentation, if you want it to be well userenced and properbose. Have fun. Your buggestions are welcome.



30. Attributes in Files and in Buffers

The LISP Machine system defines *file attributes* which are part of the contents of a text file or QFASL file and provide various information on how to process the file correctly. Some attributes are used whenever text from the file is read by the LISP reader; they specify such things as the input radix to use for integers, the package to use for LISP symbols, and the dialect of LISP syntax (Common LISP or traditional). ZMACS must refer to these attributes in order to work properly when reading LISP expressions from a ZMACS buffer. Other attributes such as the **Mode** attribute exist solely to tell ZMACS the right way to edit the file.

The attributes of a text file are recorded in the first nonblank line of the file, in text between the first and second occurrences of the string '--' on that line. If the first nonblank line does not contain such text, the file's attribute list is empty. This line is called the *attribute line*. Here is an example of one:

```
::: -- Mode:LISP; Package:ZWEI; Readtable:T; Base:8 --
```

This attribute line says that the file should be edited in LISP mode, **read** should use the readtable named **t** (traditional ZetaLISP syntax), symbols should be interned in package **ZWEI**, and integers read in octal unless the text specifies otherwise. The semicolons at the beginning are not part of the attribute list; they are present to prevent the attribute list from being taken as LISP code when the file is loaded.

A ZMACS buffer also has an attribute list, and can have an attribute line to specify it. Usually, the attributes of the buffer are the same as the attributes of the file that was visited in it, because they are based on the same text in the attribute line. However, they need not always remain the same during the editing session, as explained below.

Certain attributes are so important that ZMACS must give them values in each buffer even if they are not specified. They are the attributes **Package**, **Base** and **Readtable**. The defaults come from the variables **zwei:*default-package***, **zwei:*default-base*** and **zwei:*default-readtable***. The last two of these are set up automatically from the global values of **zwei:*read-base*** and **zwei:*readtable*** the first time you switch to a ZMACS frame after cold booting.

Here are the most common attributes:

Mode	Specifies the major mode for ZMACS to use in editing the file.
Base	Specifies the default radix for integers. While in ZMACS, the variables zwei:*read-base*

and `zwei:*print-base*` are always set to the current buffer's default radix.

- Package** Specifies the default package for symbols. While in ZMACS, `zwei:*package*` is always set to the current buffer's default package.
- Readtable** Specifies the syntax for LISP expressions. The value of the attribute is the name of a standard readtable; typically `t` for the traditional ZetaLISP syntax readtable or `common` for the Common LISP readtable. While in ZMACS, `zwei:*readtable*` is always set to the current buffer's choice of readtable.
- Lowercase** Specifies whether LISP code text in this file should be expected to be in lower case. This affects the function `electric-shift-lock-if-appropriate`. See section 23.1 [Case], page 119.
- Nofill** Specifies that Auto Fill mode should not be used on this file, even by users who turn on Auto Fill mode automatically. This should be used in files that will be edited in Text mode but whose line breaks are semantically significant. See section 24.6 [Filling], page 130.
- Fonts** Specifies the fonts to display this file in. See section 23.2 [Fonts], page 120.
- Vsp** Specifies the spacing, in pixels, between the bottom of one line and the top of the next line.
- Backspace** If non-`nil`, specifies that ZMACS should let `(OVERSTRIKE)` characters cause actual overprinting on the screen when displaying this file.
- Tab Width** Specifies the number of columns (space-widths in font A) between the tab stops used for displaying tab characters in this file.
- Patch File** If non-`nil`, specifies that it is ok for this file, when loaded into the LISP world, to redefine functions and variables defined in other files.

The attribute list of a ZMACS buffer is stored within ZMACS as a LISP property list inside the data structure (an instance) for the buffer. ZMACS commands that depend on attributes refer to this list. Editing the attribute line in the buffer does not immediately change the recorded attribute list; thus, changing the line to say `'Base: 11;'` once the buffer exists does not have any effect on the reading of integers in that buffer.

If you wish ZMACS to obey changes you have made in the text of the attribute line, use the command `M-X Reparse Attribute List`. This regenerates the LISP property list from the text, so that *all* attributes change to match the new text.

30.1 Commands Setting Buffer Attributes

The attributes of a ZMACS buffer are normally based on the file's attribute list, but sometimes it is useful to give them different values with the following commands.

M-X Set Base

M-X Set Patch File

M-X Set Fonts

M-X Set Backspace

M-X Set Vsp

M-X Set Tab Width

M-X Set Lowercase

M-X Set Nofill

Set the corresponding attribute in the current buffer.

M-X Set Package

Set the **Package** attribute. This command works a little differently from the others.

M-X Set Readtable

Similar, for the **Readtable** attribute

M-X Set Common Lisp

Another way to set the **Readtable** attribute.

The above commands, except for the last three, all work alike except for which attribute is set. The command first reads the new value of the attribute, using the minibuffer. This has the same format as it would have in the attribute line, except for **M-X Set Fonts**, where you give just font names with spaces in between. The ZMACS buffer attribute value is then set.

Finally, the command asks whether the attribute line text should be changed in the corresponding way. If you are making, or unmaking, a temporary change—such as, if you want to see how the text appears with a different tab width—you should answer **N** to this question. If you intend to affect the processing of the same file in future editing sessions, answer **Y**.

The commands **M-X Set Package**, which sets the **Package** attribute, works a little differently from the other commands. The value of the **Package** attribute must be the name of an existing package. If the argument you give is not the name of an existing package, **M-X Set Package** can create such a package to make the argument valid, but first you must type **(RETURN)** twice to exit the minibuffer; then you must confirm with 'yes'. Then comes the usual question of whether to change the text of the attribute line, and finally an unusual question: should the buffer be resectionized, so that LISP functions and variables will be known to **Meta-** in the new package instead of the old? Most of the other attributes do not affect sectionization.

M-X Set Readtable sets the **Readtable** attribute, and works like **M-X Set Package**, since the **Readtable** attribute affects sectionization and must be the name of a defined system readtable.

The two readtables normally used are the traditional ZetaLISP readtable and the Common

LISP readtable. The command **M-X Set Common Lisp** makes it easy to specify either of these two. It reads an argument in the minibuffer; the string `nil` means to use the traditional ZetaLISP readtable and any other string means to use the Common LISP readtable. The command proceeds from there like **M-X Set Readtable**.

M-X Update Attribute List alters the attribute line in the current buffer according to the current values of the buffer attributes as set previously with the other commands in this section.

31. Customization

This chapter talks about various topics relevant to adapting the behavior of ZMACS in minor ways.

31.1 Minor Modes

Minor modes are options that you can use or not. For example, Auto Fill mode is a minor mode in which Spaces break lines between words as you type. All the minor modes are independent of each other and of the selected major mode. Most minor modes say in the mode line when they are on; for example, 'Fill' in the mode line means that Auto Fill mode is on.

Append **Mode** to the name of a minor mode to get the name of a command function that turns the mode on or off. Thus, the command to enable or disable Auto Fill mode is called **M-X Auto Fill Mode**. These commands are usually invoked with **M-X**, but you can connect them to characters if you wish. With no argument, the function 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 an explicit zero argument or a negative argument always turns it off.

Some minor modes are turned on or off whenever you change to a new major mode. The variable `zwei:*unsticky-minor-modes*` contains a list of minor modes to clear, and `zwei:*initial-minor-modes*` contains a list of modes to set. In both lists, a minor mode is represented by a symbol in package **ZWEI**, such as `zwei:electric-shift-lock-mode`. By default, no minor modes are set automatically; Electric Shift Lock mode, Electric Font Lock mode and Return Indents mode are cleared.

Any newly created buffer copies some minor modes from the buffer that was current at the time of its creation. These are controlled by the variable `zwei:*transfer-minor-modes*`, whose value is a list of minor modes to copy. By default, Atom Word mode, Word Abbrev mode and Emacs mode are transferred to new buffers. Note that it is futile to transfer a minor mode that is unsticky, as it will most likely be cleared immediately when the new buffer's major mode is set up.

Auto Fill mode allows you to enter filled text without breaking lines explicitly. ZMACS inserts newlines as necessary to prevent lines from becoming too long. See section 24.6 [Filling], page 130.

Electric Shift Lock mode allows you to insert LISP code in upper case without having to use the shift key or **CAPS LOCK** key. See <undefined> [LISP Case], page <undefined>. Electric Font Lock mode automatically puts comments into font B and other text into font A. See section 23.2

[Fonts], page 120.

Atom Word mode causes the word-moving commands, in LISP mode, to move over LISP atoms instead of words. If you like to use segmented atom names like `FOOBAR-READ-IN-NEXT-INPUT-SOURCE-TO-READ`, then you might prefer not to use Atom Word mode, so that you can use `M-F` to move over just part of the atom, or `C-M-F` to move over the whole atom. If you use short names, you might like Atom Word mode. In any case, the s-expression motion commands can be used to move over atoms.

Overwrite mode causes ordinary printing characters to replace existing text instead of shoving it over. It is good for editing pictures. For example, if the point is in front of the 'B' in `'FOOBAR'`, then in Overwrite mode typing a `G` changes it to `'FOOGAR'`, instead of making it `'FOOGBAR'` as usual. Also, in Overwrite mode, `(RUBOUT)` is changed to turn the previous character into a space instead of deleting it.

Word Abbrev mode allows you to define abbreviations that automatically expand as you type them. For example, `'wam'` might expand to `'word abbrev mode'`. See chapter 28 [Abbrevs], page 175, for full information.

In Emacs mode, the characters `(ALTMODE)` and `(CONTROL-C)` are defined as prefix characters meaning "Meta" and "Control-Meta-". This is for compatibility with Emacs on ITS.

Return Indents mode exchanges the meanings of `(RETURN)` and `(LINEFEED)`. In this mode, `(LINEFEED)` just inserts a newline and `(RETURN)` indents. See chapter 22 [Indentation], page 115.

31.2 Variables

ZMACS uses many variables internally, and has others whose purpose is to be set by the user for customization. ZMACS variables are actually LISP variables, symbols in the package `ZWEI`. They can in fact be accessed or changed from LISP programs like any other variables in the LISP Machine system. But the commands discussed in this section are specifically intended for operating on the ZMACS variables intended for users to change. Most of those variables are documented in this manual, and appear in the Variables Index.

One example of such a variable is `zwei:*fill-column*`, which specifies the position of the right margin (in pixels from the left margin) to be used by the fill commands. The package prefix `zwei:` is omitted most of the time when describing variables in this manual, since it applies to all of the ZMACS variables.

M-X Describe Variable

Print the value and documentation of a ZMACS variable.

M-X Set Variable

Change the value of a variable.

M-X List Variables

Print a list of all ZMACS variables.

M-X Variable Apropos

Print a list of some ZMACS variables.

M-X Make Local Variable

Make a ZMACS variable have a local value in the current buffer.

M-X Kill Local Variable

Make a ZMACS variable use its global value in the current buffer.

M-X List Local Variables

List the ZMACS variables local in the current buffer.

ZMACS variables, following the convention for LISP variables, contain all upper case letters that can be written in LISP code either in upper or lower case, and that appear in lower case in documentation (example: `zwei:*fill-column*`). The ZMACS commands for managing variables refer to the name in a different syntactic form, obtained by removing the asterisks, converting hyphens to spaces, and upcasing the first letter of each word (example: `Fill Column`).

To examine the value of a single variable, use **M-X Describe Variable**, which reads a variable name using the minibuffer, with completion. It prints both the value and the documentation of the variable.

M-X Describe Variable `(RETURN)` `Fill Column` `(RETURN)`

prints something like

```
Fill Column:      576 pixels (72 spaces in current font, plus 0 pixels)
Width in pixels used for filling text.
```

The easiest way for the beginner to set a named variable is to use **M-X Set Variable**. This reads the variable name with the minibuffer, prints a description just like **M-X Describe Variable**, and then reads a new value using the minibuffer. When the value is read, the minibuffer starts out containing the text for the current value. This makes it easy to set the variable to a slightly different value. If you want to specify a completely different value, use **C-K** to kill the text initially

supplied. For example,

```
M-X Set Variable (RETURN) Fill Column (RETURN) C-K 576 (RETURN)
```

sets `zwei:*fill-column*` to 576.

LISP variables are *untyped*—any variable may be given any LISP object as its value—but ZMACS knows which type of value each of its advertised variables should have. **M-X Set Variable** describes the expected type in the prompt when it asks for the new value, and it checks the value for validity.

Some ZMACS variables have lists of characters as their values—actual LISP lists, like `(#/space #/tab)`. The commands in this section print their values as strings, and **Set Variable** expects string-like syntax for the new value.

To print a complete list of all variables, do **M-X List Variables**. With a numeric argument, it prints the documentation of each variable, as well as the name and the value. To print only some variables, use **M-X Variable Apropos**, which reads a string argument and prints only variables whose names contain that string. (Extended search characters are allowed; see section 15.7 [Extended Search], page 68)

If you want to set a variable a particular way each time you use ZMACS, you can use the LISP function `setq` in your `LISPM.INIT` file.

Any variable can be made *local* to a specific ZMACS buffer. This means that its value in that buffer is independent of its value in other buffers. The variables that major modes typically set are always local in each ZMACS buffer; this is why changing major modes in one buffer has no effect on other buffers. Every other ZMACS variable has a *global* value which is in effect in all buffers that have not made the variable local.

M-X Make Local Variable reads the name of a variable and makes it local to the current buffer. Further changes in this buffer will not affect others, and further changes in the global value will not affect this buffer.

M-X Kill Local Variable reads the name of a variable and makes it cease to be local to the current buffer. The global value of the variable henceforth is in effect in this buffer.

M-X List Local Variables prints a list of all variables that have been made local to the

current buffer using **M-X Make Local Variable**. This does not include the few variables that are always local in every buffer.

31.3 Keyboard Macros

A *keyboard macro* is a command defined by the user to abbreviate a sequence of other commands. For example, if you discover that you are about to type **C-N C-D** forty times, you can speed your work by defining a keyboard macro to do **C-N C-D** and calling it with a repeat count of forty.

C-X (Start defining a keyboard macro.

C-X) End the definition of a keyboard macro.

C-X E Execute the most recent keyboard macro.

C-U C-X (
Re-execute last keyboard macro, then add more commands to its definition.

C-X Q Ask for confirmation when the keyboard macro is executed.

M-X Name Last Kbd Macro

Give a permanent name (for the duration of the session) to the most recently defined keyboard macro.

(MACRO) C Call macro specified by name.

M-X Install Macro

Connect a sequence of command characters to a keyboard macro.

M-X Install Mouse Macro

Connect a sequence of command characters (usually a mouse click character) to a keyboard macro, and arrange to move point to the mouse position before executing the macro.

M-X View Kbd Macro

Print the sequence of command characters that make up the definition of a keyboard macro.

Keyboard macros differ from ordinary ZMACS commands, in that they are written in the ZMACS command language rather than in LISP. This makes it easier for the novice to write them, and makes them more convenient as temporary hacks. However, the ZMACS command language is not powerful enough as a programming language to be useful for writing anything intelligent or general. For such things, LISP must be used.

You define a keyboard macro while executing the commands that are the definition. Put differently, as you are defining a keyboard macro, the definition is being executed for the first

time. This way, you can see what the effects of your commands are, so that you don't have to figure them out in your head. 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.

31.3.1 Basic Use

To start defining a keyboard macro, type the **C-X (command (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 to remind you of what is going on. When you are finished, the **C-X) command (End Kbd Macro)** terminates the definition (without becoming part of it!). For example

```
C-X ( M-F foo C-X )
```

defines a macro to move forward a word and then insert **'foo'**.

The macro thus defined can be invoked again with the **C-X E command (Call Last Kbd Macro)**, which may be given a repeat count as a numeric argument to execute the macro many times. **C-X)** can also be given a repeat count as an 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). So, giving **C-X)** an argument of 4 executes the macro immediately 3 additional times. An argument of zero to **C-X E** or **C-X)** means repeat the macro indefinitely (until it gets an error, or you type **C-ABORT**).

If you wish to repeat an operation at regularly spaced places in the text, define a macro and include as part of the macro the commands to move to the next place you want to use it. For example, if you want to change each line, you should position point at the start of a line, and define a macro to change that line and leave point at the start of the next line. Then repeating the macro will operate on successive lines.

After you have terminated the definition of a keyboard macro, you can add to the end of its definition by typing **C-U C-X (**. This is equivalent to plain **C-X (** followed by retyping the whole definition so far. As a consequence it re-executes the macro as previously defined.

To examine the definition of a keyboard macro, use **M-X View Kbd Macro**. Supply as an argument the name previously given to the macro with **M-X Name Last Kbd Macro**, or supply an empty argument (type just **RETURN**) to view the last macro defined.

31.3.2 Naming and Installing Keyboard Macros

If you wish to save a keyboard macro for longer than until you define the next one, you must give it a name or install it on a command sequence. To give the macro a name, use **M-X Name Last Kbd Macro**. This reads a name as an argument using the minibuffer and defines that name to execute the macro. Names for macros look like command function names—words separated by spaces—but are a separate name space; use of a name for a macro does not conflict with its use as the name of a command function.

Another way to define and name a macro is to type **(MACRO) M name (RETURN)**. This is like **C-X** (except that the macro automatically gets the name *name* as soon as you finish defining it).

To call the macro named *name*, type **(MACRO) C name (RETURN)**.

Installing a keyboard macro on a command sequence makes it most convenient to use. To do this, use **M-X Install Macro**. You must answer several questions:

1. Give the name (assigned using **M-X Name Last Kbd Macro**) of the macro you want to use, or supply an empty argument to use the last macro defined.
2. Type the character command (or sequence starting with **C-X**) that you wish to redefine. Sequences starting with **M-X** are not handled.
3. Confirm the macro name and character sequence with **Y**. Take this opportunity to check that the command sequence being defined is correct. If you redefine an important character such as **(RUBOUT)** or **M-X**, further use of ZMACS can be very painful.
4. Specify how globally the character should be redefined: either **W** for just in this window (this ZMACS frame), **Z** for every ZMACS frame, or **A** for all ZWEI-based editors including ZMail and Converse.

For example,

```
M-X Install Macro (RETURN) (RETURN) Hyper-Q Y Z
```

connects the last macro defined to the character **Hyper-Q** in every ZMACS frame.

Install Macro actually works just like **Define Character** except that you specify a defined macro name rather than a command function name.

Mouse clicks such as \mathbb{M} or **Meta- \mathbb{M}** can be defined to run macros using **Install Macro**, but unless the macro runs a command that looks at the mouse position (which is unlikely), its execution will be based on the location of point at the time the mouse is clicked and not on the mouse cursor location. However, the command **M-X Install Mouse Macro** can also be used. It works like **M-X Install Macro** except that it arranges specially for point to move to the mouse position before the macro definition is executed.

You can define a keyboard character with **Install Mouse Macro**, if you want to have a keyboard command whose meaning depends on the position of the mouse.

31.3.3 Nesting Macro Definitions

You can define a macro within another macro; this is useful for making nested loops of commands. The inner macro will be defined and then repeatedly used on each iteration of the outer macro. To do this, just use the **C-X** (and **C-X**) commands within the definition of a macro. For example,

```
C-X ( Foo C-X (  $\langle$ SPACE $\rangle$  bar C-U 10 C-X ) .  $\langle$ RETURN $\rangle$  C-U 5 C-X )
```

defines an inner macro repeated ten times, to insert ten copies of ' bar', for each time through the outer macro. The result is to insert

```
Foo bar bar bar bar bar bar bar bar bar bar.
Foo bar bar bar bar bar bar bar bar bar bar.
Foo bar bar bar bar bar bar bar bar bar bar.
Foo bar bar bar bar bar bar bar bar bar bar.
Foo bar bar bar bar bar bar bar bar bar bar.
```

It is also possible to call a macro by name or by a character command in the definition of another macro.

31.3.4 Executing Macros with Variations

Using **C-X Q** (**Kbd Macro Query**), you can get an effect similar to that of **Query Replace**, where the macro asks you each time around whether to make a change. When you are defining the macro, type **C-X Q** at the point where you want the query to occur. During macro definition, the

C-X Q does nothing, but when the macro is invoked the **C-X Q** reads a character from the terminal to decide whether to continue.

The special answers are **(SPACE)**, **(RUBOUT)**, **(CLEAR-SCREEN)**, and **C-R**. Any other character terminates execution of the keyboard macro and is then read as a command. **(SPACE)** means to continue. **(RUBOUT)** means to skip the remainder of this repetition of the macro, starting again from the beginning in the next repetition. **(CLEAR-SCREEN)** clears the screen and asks you again for a character to say what to do. **C-R** enters a recursive editing level, in which you can perform different editing each time through the macro. When you exit the recursive edit using **(END)**, you are asked again how to continue with the keyboard macro. If you type a **(SPACE)**, the rest of the macro definition is executed. It is up to you to leave point and the text in a state such that the rest of the macro will do what you want.

31.4 Command Functions and Command Tables

This section deals with the *comtabs* which define the connections between character commands and command functions, and say how you can customize these connections.

31.4.1 Command Functions

A command functions is a LISP function with a few extra pieces of information. Like every LISP function, a command function has a LISP function name, a LISP symbol whose name usually contains upper case letters and dashes, and starts (by convention) with the word **com**. Command functions also have command names, as explained above. The command name is normally computed from the LISP function name by removing the word **com**, changing dashes to spaces, and capitalizing each word. Thus, the function name **com-down-real-line** gives the command name **Down Real Line**.

Command functions are defined with a special macro **zwei:defcom** which computes a command name from the function name and records it on the property list of the function name.

31.4.2 Changing Comtabs from ZMACS

The connections between characters and command functions are recorded in data structures called *comtabs* or *command tables*. Each ZMACS frame has its own comtab. There is also one comtab shared by all ZMACS frames; the individual frame's comtabs all inherit from this one.

Yet another comtab is common to all ZWEI-based editors (see section 2.4 [ZWEI], page 13). The ZMACS comtab inherits definitions from this one. (The whole truth is more complicated than this, but this explains the effects you see from the commands described here.)

To alter the comtab of the current ZMACS frame, use **M-X Define Character**. You must answer several questions:

1. Give the command name of the command function you want to use. Completion is available.
2. Type the character command (or sequence starting with **C-X**) that you wish to redefine. Sequences starting with **M-X** are not handled.
3. Confirm the macro name and character sequence with **Y**. Take this opportunity to check that the command sequence being defined is correct. If you redefine an important character such as **(RUBOUT)** or **M-X**, further use of ZMACS can be very painful.
4. Specify how globally the character should be redefined: either **W** for just in this window (this ZMACS frame), **Z** for every ZMACS frame, or **A** for all ZWEI-based editors including ZMail and Converse.

For example,

M-X Define Character (RETURN) Forward Word (RETURN) Super-F Y W

would define **Super-F** to move forward a word, like **Meta-F**, in this ZMACS frame only.

Mouse commands can also be defined this way. The **(CONTROL)**, **(META)**, **(SUPER)** and **(HYPER)** keys can be used with the mouse click to increase the number of different mouse commands you can have. For example,

M-X Define Character (RETURN) Forward Word (RETURN) Meta-{L} Y (Z)

defines clicking **(L)** with **(CONTROL)** held down to move the cursor forward a word, in all ZMACS frames. Generally the mouse position is not used by such commands. For example, **Forward Word** moves by words starting from the previous position of point; it does this when invoked by **M-F**, and it does the same thing if invoked by **Meta-(L)**. The command functions used to define the standard ZMACS mouse commands look at the mouse position because they are specially programmed to do so.

M-X Define Character can be used to redefine characters already defined. To make a character undefined, use **M-X Undefine Character**.

An alternative interface is **M-X Install Command**. It works like **M-X Define Character**, except that it expects the name of a LISP function rather than a ZMACS command function name. For example,

```
M-X Install Command RETURN zwei:com-forward-word RETURN Super-F
```

has the same effect as the previous example.

The command **M-X Install Macro** also modifies the comtab. See section 31.3 [Keyboard Macros], page 195.

In LISP code, such as in LISP.MINIT files, the way to modify the comtab is to call **zwei:set-comtab**. Refer to the self-documentation of that function.

31.4.3 Meta-X Availability

Not all command functions in ZMACS can be invoked with **Meta-X** at any time. The comtabs that connect characters to command functions also specify exactly which command functions you can call using **Meta-X**. There are three possible reasons why a command might not be made available for **M-X**:

1. It is connected to a one or two character command. With a few exceptions, command functions connected to characters are not normally made available through **Meta-X**. This is to make command completion work better.
2. It is experimental, or we are not sure it is worth recommending
3. It is not meaningful to use it in the current context. For example, it may be one of the commands designed for use in **Dired**, which assume that the current buffer is a **Dired** buffer. In **Dired**, these commands are connected to characters. Outside of **Dired**, they are meaningless and could even cause trouble if they were used.

Occasionally it is useful to invoke a command by name which is not on the **Meta-X** lists in the current comtabs. This can be done using **Control-Meta-X (Any Extended Command)**. This is used just like **Meta-X**, except that it ignores the comtabs; it accepts any ZMACS command name.

31.5 The Syntax Table

All the ZMACS commands that parse words or balance parentheses are controlled by the *syntax tables*. There are two syntax tables, one defining the *word syntax* of each character and one defining the *list syntax* of each character.

By changing the word syntax, you can control whether a character is considered a word delimiter or part of a word. By changing the list syntax, you can control which characters are parentheses, which ones are parts of symbols, which ones are prefix operators, and which ones are just ignored when parsing *s*-expressions.

Each buffer has its own pair of syntax tables, and changes in the syntax made in one buffer (such as when a new major mode is selected) do not affect other buffers.

The syntax tables are not always stored as vectors of length 256, but you can understand their functionality by thinking of them that way.

The word syntax of a character has only two possible alternatives: **word-alphabetic** and **word-delimiter**. These are constants whose values are zero and one. The list syntax of a character has several alternative values, all small integers. These symbols are constants whose values are the numbers found in the syntax table:

list-alphabetic

This character is a symbol constituent.

list-delimiter

This character separates things but has no other significance.

list-slash This character quotes the following character.

list-double-quote

This character starts a grouping terminated by another of the same character.

list-single-quote

This character is part of whatever expression follows it.

list-close This character acts like a close parenthesis.

list-open This character acts like an open parenthesis.

list-comment

This character starts a comment.

list-colon This character ends a package prefix.

When a character has the syntax of an open parenthesis, that means that the character is taken

to be the beginning of a parenthesized grouping when expressions are being parsed. Thus, any number of different expression-starting characters can be handled.

The syntax **list-single-quote** means that the character becomes part of whatever object follows it, whether symbol or list, and can also be in the middle of a symbol, but does not constitute anything by itself if surrounded by whitespace.

A character of syntax **list-slash** causes itself and the next character to be treated as alphabetic.

A string quote is one that matches in pairs. All characters inside a pair of string quotes are treated as alphabetic except for the character quote, which retains its significance, and can be used to force a string quote or character quote into a string.

A comment starter is taken to start a comment, which ends at the end of the line, suppressing the normal syntax of all characters between. Only the list and s-expression commands use the syntax table to find comments; the commands specifically for comments have other variables that tell them where to find comments.



32. Correcting Mistakes and ZMACS Problems

If you type an ZMACS command you did not intend, the results are often mysterious. This chapter tells what you can do to cancel your mistake or recover from a mysterious situation. ZMACS bugs and system crashes are also considered.

32.1 Quitting and Aborting

- C-G** Abort one stage. Cancels a partially typed command, or erases minibuffer text, or gets out of the minibuffer, but never more than one of these at a time.
- (ABORT)** Full abort. Cancels any partially typed command, and gets out of the minibuffer if in one. Also gets out of recursive edits.
- C-(ABORT)** Stop an executing command instantly. Does not exit minibuffers or recursive edits.

There are three ways of cancelling commands that are not finished executing: *partial abort* with **C-G**, *full abort* with **(ABORT)**, and instant stop with **C-(ABORT)**.

Partial abort with **C-G** can cancel several kinds of things. Each use of **C-G** performs one action, the first applicable one, from the following table.

partial command

Prefix characters or numeric arguments constitute partially-typed commands. **C-G** cancels all of them.

incremental search, not successful

In an incremental search that has not finished searching for all the argument characters supplied, or that has failed to find them, **C-G** cancels characters in the search argument that have not been found.

incremental search, successful

In an incremental search that has found all the specified search string, **C-G** cancels the search: it moves point back to the starting point of the search and terminates searching.

region If a region is active, **C-G** makes it inactive.

minibuffer text

When in the minibuffer and the minibuffer is not empty, **C-G** deletes the text in it.

empty minibuffer

In an empty minibuffer, **C-G** exits the minibuffer and cancels the command that was using the minibuffer to read an argument.

recursive edit

Within a recursive edit (such as inside **Edit Tab Stops**), **C-G** throws back to top level, cancelling the command for which the recursive edit was being done.

C-G does only one of these things each time it is used: if you type **C-U 5 C-X** in a minibuffer containing text and with an active region, it takes four **C-G** commands to get back to top level. The first one cancels the argument and **C-X**; the second cancels the region, the third deletes the text, and the fourth gets out of the minibuffer.

Full abort with **(ABORT)** is like several **C-G** commands. It performs *all* the applicable actions from the above table, one by one, until top level is reached. Only one **(ABORT)** is needed to reach top level.

C-(ABORT) for aborting a running program instantly is a standard LISP Machine feature, available also in ZMACS. It cancels an executing ZMACS command, but does not exit from any command level (minibuffer or recursive edit).

32.2 Dealing with ZMACS Trouble

This section describes various conditions that can cause ZMACS not to work, or cause it to display strange things, and how you can correct them.

32.2.1 Subsystems and Recursive Editing Levels

Recursive editing levels are important and useful features of ZMACS, but they can seem like malfunctions to the user who does not understand them.

If the mode line starts with a bracket '[' , you have entered a recursive editing level. To get back to top level, type **(ABORT)**.

32.2.2 Garbage on the Screen

If the data on the screen looks wrong, the first thing to do is see whether the text is really wrong. Type **(CLEAR-SCREEN)**, to redisplay the entire screen. If it appears correct after this, the problem was entirely in the previous screen bit map.

32.2.3 Garbage in the Text

If **CLEAR-SCREEN** shows that the text is wrong, try undoing the changes to it using **C-Shift-U** until it gets back to a state you consider correct.

32.2.4 ZMACS Hung and Not Responding

If ZMACS does not respond to commands, type **SYSTEM C-E**. If the LISP Machine itself is still running, this makes and selects a fresh ZMACS frame. Just continue editing with the new one and ignore the old one. The new ZMACS frame will initially show a new empty buffer, but you can switch to any of the old buffers in it.



Glossary

Abbrev An abbrev is a text string which expands into a different text string when present in the buffer. For example, you might define a short word as an abbrev for a long phrase that you want to insert frequently. See chapter 28 [Abbrevs], page 175.

Aborting Aborting means canceling a partially typed command or getting out of a temporary special situation (such as a minibuffer (q.v.) or recursive edit (q.v.)). The commands **ABORT** and **C-G** are used for this. See section 32.1 [Quitting], page 205.

Accumulator

Some of the special Zmacs commands for editing assembler language code are designed to operate on the accumulator field of an instruction. See section 29.5 [MIDAS], page 183.

Activation of the Mark

When an active mark is set, an underlined region exists, and commands can operate on the region. When the mark is not active, nothing is underlined, and no command operates on the region; commands that are defined to require a region are not valid then. See chapter 10 [Mark], page 39.

Activation in Ztop

Activation in Ztop means telling Ztop to allow the input already inserted to be read by the program. If the I-blinker is visible, it means that Ztop input is now dormant (q.v.), and will not be processed unless you activate. See chapter 27 [Ztop], page 171.

Address Some of the special Zmacs commands for editing assembler language code are designed to operate on the address field of an instruction. See section 29.5 [MIDAS], page 183.

ALTMODE **ALTMODE** is a character, used to end incremental searches and to request completion (q.v.) in the minibuffer (q.v.).

Atom Word mode

Atom Word mode is a minor mode. When it is enabled, the Zmacs commands for operating on words treat a whole Lisp atom as a word. See section 31.1 [Minor Modes], page 191.

Attribute Line

The attribute line in a file or buffer is the first nonblank line, provided that it contains the string '-*-'. This line specifies the attributes (q.v.) of the file or buffer. See chapter 30 [Attributes], page 187.

Attributes

Attributes are properties belonging to the contents of a file or buffer, saying how to edit, load, compile or print the text. For example, the **Fonts** attribute says which fonts to use to display the text in Zmacs or to print it on a hardcopy device. See chapter 30 [Attributes], page 187.

Auto Fill mode

Auto Fill mode is a minor mode in which text that you insert is automatically broken into lines of fixed width. See section 24.6 [Filling], page 130.

Balance Parentheses

EMACS can balance parentheses manually or automatically. Manual balancing is done by the commands to move over balanced expressions (see section 25.2 [Lists], page 136). Automatic balancing is done by blinking the parenthesis that matches one next to point (see section 25.5 [Matching Parens], page 143).

Blank Lines

Blank lines are lines that contain only whitespace. Zmacs has several commands for operating on the blank lines in the buffer.

Buffer

The buffer is the basic editing unit; one buffer corresponds to one piece of text being edited. You can have several buffers, but at any time you are editing only one, the 'selected' buffer, though two can be visible when you are using two windows. See chapter 19 [Buffers], page 99.

Buffer Group

A buffer group is an ordered collection of buffers which is specified to Zmacs so that you can search or replace through all of them with a single command. See section 19.6 [Buffer Groups], page 104.

Buffer Selection History

Each Zmacs window has a buffer selection history which records how recently each Zmacs buffer has been selected in that window. Each window's buffer selection history contains all the Zmacs buffers there are, but the ordering of the buffers differs from window to window. See chapter 19 [Buffers], page 99.

C-

'C' in the name of a character is an abbreviation for Control. See chapter 2 [Characters], page 11.

C-M-

'C-M-' in the name of a character is an abbreviation for Control-Meta. See chapter 2 [Characters], page 11.

Case Conversion

Case conversion means changing text from upper case to lower case or vice versa. See section 23.1 [Case], page 119, for the general purpose commands for case conversion. See section 25.7 [Lisp Case], page 146, for commands for case conversion of Lisp code in particular.

Changed Sections

A section is changed if its text has been changed since some previous time. Various Zmacs commands operate on only the changed sections of one or more buffers. See section 26.1 [Sectionization], page 153.

Characters

Most Zmacs commands are single characters that take effect immediately. See chap-

ter 2 [Characters], page 11.

Command

A command is a character or sequence of characters which, when typed by the user, fully specifies one action to be performed by EMACS. For example, **X** and **Control-F** and **Meta-X Text Mode** (**RETURN**) are commands. See section 2.3 [Commands], page 12. Sometimes the first character of a multi-character command is also considered a command in its own right: **M-X Text Mode** (**RETURN**) is a command (an extended command), and **M-X** is also a command (a command to read a function name and invoke the function). See chapter 8 [Extended Commands], page 35.

Command Function

A command function is a Lisp function specially arranged to be able to serve as a Zmacs command definition. When you type a character command in Zmacs, it is looked up in the comtab (q.v.) to find the command function it is connected to; then that command function is executed to produce the editing effect of the character you typed. See section 2.3 [Commands], page 12.

Command Name

A command name is the name assigned by Zmacs to a command function. In an extended command, you invoke a command function by typing the command name. See section 2.3 [Commands], page 12.

Command Table

See "Comtab".

Comments

A comment is text in a program which is intended only for humans reading the program, and is marked specially so that it will be ignored when the program is loaded or compiled. EMACS offers special commands for creating and killing comments. See section 25.6 [Comments], page 143.

Compilation

When you edit a Lisp program with Zmacs, you can compile the changed code from the Zmacs buffer. See section 26.3 [Compile Text], page 157. You can also compile from source files with Zmacs (see section 26.2 [Compile File], page 157).

Completion

Completion is what EMACS does when it automatically fills out an abbreviation for a name into the entire name. Completion is done for minibuffer (q.v.) arguments, when the set of possible valid inputs is known; for example, on extended command names, buffer names, and file names. Completion occurs when (**ALTMODE**), (**SPACE**) or (**RETURN**) is typed. See section 7.1 [Completion], page 30.

Comtab The comtab is the data structure that records the connections between command characters and the command functions that they run. For example, the comtab connects the character **C-N** to the command function **Down Real Line**. See section 31.4

[Comtabs], page 199.

Connected

A one- or two-character command in EMACS works by calling a command function (q.v.) which it is *connected to*. Customization often involves connecting a character to a different command function. The connections are recorded in the comtab (q.v.). See section 2.3 [Commands], page 12.

Continuation Line

When a line of text is longer than the width of the screen, it takes up more than one screen line when displayed. We say that the text line is continued, and all screen lines used for it after the first are called continuation lines. See chapter 4 [Basic Editing], page 19.

Control Control is the name of a modifier bit which a command character may have. It is present in a character if the character is typed with the **CONTROL** key held down. Such characters are given names that start with **Control-**. For example, **Control-A** is typed by holding down **CONTROL** and typing **A**. See chapter 2 [Characters], page 11.

Control-Character

A Control character is a character which includes the Control bit.

Control-X Command

A Control-X command is a two-character command whose first character is the prefix character Control-X. See section 2.2 [Multicharacter Commands], page 11.

Current Buffer

The current buffer in Zmacs is the Zmacs buffer on which most editing commands operate. You can select any Zmacs buffer as the current one. See chapter 19 [Buffers], page 99.

Current Font

When editing a multi-font buffer, the current font is the one in which newly inserted text will go.

Current Line

The line point is on.

Current Paragraph

The paragraph that point is in. If point is between paragraphs, the current paragraph is the one that follows point. See section 24.4 [Paragraphs], page 128.

Current Defun

The defun that point is in. If point is between defuns, the current defun is the one that follows point. See section 25.3 [Defuns], page 138.

Cursor The cursor is the rectangle on the screen which indicates the position called point (q.v.) at which insertion and deletion takes place. The cursor is part of the window system, which Zmacs uses to do its display. Often people speak of "the cursor" when, strictly speaking, they mean "point". See chapter 4 [Basic Editing], page 19.

Customization

Customization is making minor changes in the way EMACS works. It is often done by setting variables (see section 31.2 [Variables], page 192) or by reconnecting commands (see section 31.4 [Comtabs], page 199).

Default Argument

The default for an argument is the value that will be assumed if you do not specify one. When the minibuffer is used to read an argument, the default argument is used if you just type **RETURN**. See chapter 7 [Minibuffer], page 29.

Default File Name

When a file name arguments is specified, any file name components that you do not specify are taken from the corresponding components of the default file name. In Zmacs, the default file name is normally the name of the file visited in the current buffer. See the chapter on Pathnames in the *Lisp Machine Manual* for more information on file name defaulting.

Defun A defun is a list at the top level of list structure in a Lisp program. It is so named because most such lists are calls to the Lisp function defun. See section 25.3 [Defuns], page 138.

Deletion Deletion means erasing text without saving it. EMACS deletes text only when it is expected not to be worth saving (all whitespace, or only one character). The alternative is killing (q.v.). See section 11.1 [Killing], page 45.

Deletion of Files

Deletion of a file means erasing it from the file system. See section 17.6 [Deleting Files], page 82.

Directory

Files in a file server are grouped into file directories. See section 17.5 [Directories], page 81.

Dired Dired is the Zmacs facility that displays the contents of a file directory and allows you to "edit the directory", performing operations on the files in the directory. See chapter 18 [Dired], page 91.

Dormant When Ztop input is dormant, the program is not allowed to read it. This happens if you move point away from the end of the Ztop buffer. The program can read input again if you activate input (q.v.). See chapter 27 [Ztop], page 171.

Echo Area

The echo area is the bottom three lines of the screen, used for echoing the arguments to commands, for asking questions, and printing brief messages (including error messages). See section 1.3 [Echo Area], page 6.

Echoing Echoing is acknowledging the receipt of commands by displaying them (in the echo area). EMACS never echoes single-character commands; longer commands echo only if you pause while typing them.

Electric Shift Lock Mode

Electric Shift Lock mode is a minor mode in which letters in Lisp code are automatically converted to upper case when not quoted and not in strings or comments. See section 25.7 [Lisp Case], page 146.

Electric Font Lock Mode

Electric Font Lock mode is a minor mode in which Lisp comments are automatically put into font B, and the actual Lisp program into font A. See section 23.2 [Fonts], page 120.

Error Messages

Error messages are single lines of output printed by Zmacs when the user asks for something impossible to do (such as, killing text forward when point is at the end of the buffer). They appear in the echo area, accompanied by a beep.

Exchanging

See “transposition”.

Expunging

Expunging is an operation performed on file directories. On some file servers, deleted (q.v.) files are not really erased until the directory (q.v.) containing them is expunged. See section 17.6 [Deleting Files], page 82.

Extended Command

An extended command is a command which consists of the character **Meta-X** followed by the command name (really, the name of a command function (q.v.)). An extended command requires several characters of input, but its name is made up of English words, so it is easy to remember. See chapter 8 [Extended Commands], page 35.

Extended Search

Extended search characters let you specify a pattern rather than an exact string as the target to search for in certain Zmacs commands. See section 15.7 [Extended Search], page 68.

File Attributes

See “Attributes”.

File Directory

See “Directory”.

File Properties

File properties are information attached to a file by the file server, aside from the text of the file. Different file servers implement different properties; typically they include the creation date and the name of the file’s author. File properties in general are discussed in the section “Accessing Directories” in the *Lisp Machine Manual*. See section 18.4 [Dired Props], page 93, for Zmacs commands for working with file properties.

Fill Prefix

The fill prefix is a string that should be expected at the beginning of each line when

filling is done. It is not regarded as part of the text to be filled. See section 24.6 [Filling], page 130.

Filling Filling text means moving text from line to line so that all the lines are approximately the same length. See section 24.6 [Filling], page 130.

Fonts A font is a set of images that can be used to display characters of text. A single Zmacs buffer can use several fonts to display the characters in it. See section 23.2 [Fonts], page 120.

Font Attribute

The **Font** attribute of a buffer or file says which fonts to use for displaying that buffer or file. See chapter 30 [Attributes], page 187. See also "attributes".

Global The global value of a variable (q.v.) applies to all buffers except those which have their own local values of the variable. See section 31.2 [Variables], page 192.

Global Substitution

Global substitution means replacing one string by another string through a large amount of text. See section 15.5 [Replace], page 65.

Graphic Character

Graphic characters are those assigned pictorial images rather than just names. These include letters, digits, punctuation, and spaces; they do not include **(RETURN)** or **(RUBOUT)**. In Zmacs, typing graphic characters inserts those characters. See chapter 4 [Basic Editing], page 19.

Grinding Grinding means reformatting a program so that it is indented according to its structure. See chapter 22 [Indentation], page 115.

H- **H-** in the name of a character is an abbreviation for **(HYPER)**, one of the modifier keys that can accompany any character. See chapter 2 [Characters], page 11.

Hardcopy

Hardcopy means printed output. Zmacs has commands for making printed listings of files or of text in Zmacs buffers. See section 29.3 [Hardcopy], page 180.

(HELP) You can type the **(HELP)** character at any time to ask what options you have, or to ask what any command does. See chapter 9 [Help], page 37.

Hyper Hyper is the name of a modifier bit which a command character may have. It is present in a character if the character is typed with the **(HYPER)** key held down. Such characters are given names that start with **Hyper-**. For example, **Hyper-Q** is typed by holding down **(HYPER)** and typing **Q**. See chapter 2 [Characters], page 11.

Indentation

Indentation means blank space at the beginning of a line. Lisp and other programming languages have conventions for using indentation to illuminate the structure of the program, and Zmacs has special features to help you set up the correct indentation. See chapter 22 [Indentation], page 115.

Insertion Insertion means copying text into the buffer, either from the keyboard or from some other place in the Lisp Machine.

Justification

Justification means adding extra spaces to lines of text to make them come exactly to a specified width. See section 24.6 [Filling], page 130.

Keyboard Macros

Keyboard macros are a way of defining new Zmacs commands from sequences of existing ones, with no need to write a Lisp program. See section 31.3 [Keyboard Macros], page 195.

Kill History

The kill history is where all text you have killed is saved. You can reinsert any of the killed text for the rest of the session; this is called yanking (q.v.). See section 11.2 [Yanking], page 47.

Killing Killing means erasing text and saving it on the kill history so it can be yanked (q.v.) later. Most EMACS commands to erase text do killing, as opposed to deletion (q.v.). See section 11.1 [Killing], page 45.

Label A label is a line of text inside the top or borrom edge of a window saying what the window is for. Zmacs windows have labels when there is more than one ina Zmacs frame. See chapter 20 [Windows], page 109.

List A list is, approximately, a text string beginning with an open parenthesis and ending with the matching close parenthesis. Zmacs has special commands for many operations on lists, because they are useful in editing Lisp code. See section 25.2 [Lists], page 136.

Local Variable

A local value of a variable (q.v.) applies to only one buffer. See section 31.2 [Variables], page 192.

M- **M-** in the name of a character is an abbreviation for **(META)**, one of the modifier keys that can accompany any character. See chapter 2 [Characters], page 11.

M-X **M-X** is the character which begins an extended command (q.v.). Extended commands have come to be known also as "M-X commands", and an individual extended command is often referred to as "M-X such-and such". See chapter 8 [M-X], page 35.

Mail Mail means messages sent from one user to another through the computer system. Zmacs has commands for composing and sending mail, but to read mail you must use ZMail. See section 29.2 [Mail], page 179.

Major Mode

The major modes are a mutually exclusive set of options each of which configures EMACS for editing a certain sort of text. Ideally, each programming language has its own major mode. See chapter 21 [Major Modes], page 113.

Mark The mark points to a position in the text. It specifies one end of the region (q.v.), point being the other end. Many commands operate on all the text from point to the

mark. See chapter 10 [Mark], page 39.

Message See "mail".

Meta Meta is the name of a modifier bit which a command character may have. It is present in a character if the character is typed with the **(META)** key held down. Such characters are given names that start with **Meta-**. For example, **Meta-<** is typed by holding down **(META)** and typing **<** (which itself is done by holding down **(SHIFT)** and typing **.**). See chapter 2 [Characters], page 11.

Meta Character

A Meta character is one whose character code includes the Meta bit.

Microcompilation

Microcompilation is compiling Lisp code into Lisp Machine microcode. It is mentioned here because Zmacs has a command to microcompile a Lisp function. See section 26.3 [Compile Text], page 157.

Minibuffer

The minibuffer is the window that appears when necessary inside the echo area (q.v.), used for reading arguments to commands. See chapter 7 [Minibuffer], page 29.

Minor mode

A minor mode is an optional feature of EMACS which can be switched on or off independently of all other features. Each minor mode has an extended command to turn it on or off. See section 31.1 [Minor Modes], page 191.

Mode line

The mode line is the line just above the echo area (q.v.), used for status information. See section 1.4 [Mode Line], page 7.

Modified Buffer

A buffer is modified if its text has been changed since the last time the buffer was saved (or when it was created, if it has never been saved).

Moving Text

Moving text means erasing it from one place and inserting it in another. This is done by killing (q.v.) and then yanking (q.v.).

Named Mark

A named mark is a register (q.v.) in its role of recording a location in text so that you can move point to that location. See section 10.4 [Named Marks], page 42.

Newline **(RETURN)** characters (q.v.) terminating lines of text are called newlines. See chapter 2 [Characters], page 11.

Numeric Argument

A numeric argument is a number, specified before a command, to change the effect of the command. Often the numeric argument serves as a repeat count. See chapter 6 [Arguments], page 27.

Overwrite Mode

Overwrite mode is a minor mode. When it is enabled, ordinary text characters replace the existing text after point rather than pushing it to the right.

Page A page is a unit of text, delimited by `<PAGE>` characters coming at the beginning of a line. Some Zmacs commands are provided for moving over and operating on pages. See section 24.5 [Pages], page 129.

Paragraphs

Paragraphs are the medium-size unit of English text. There are special Zmacs commands for moving over and operating on paragraphs. See section 24.4 [Paragraphs], page 128.

Parsing We say that EMACS parses words or expressions in the text being edited. Really, all it knows how to do is find the other end of a word or expression. See section 31.5 [Syntax], page 201.

Patches Patches are files containing corrections to previous compiled versions of large Lisp programs, made to be loaded in on top of the old version in order to bring it up to date. The section "The Patch Facility" in the *Lisp Machine Manual* discusses patches thoroughly. See section 26.8 [Patches], page 165, for information on the Zmacs commands used to create patch files.

Point Point is the place in the buffer at which insertion and deletion occur. Point is considered to be between two characters, not at one character. The terminal's cursor (q.v.) indicates the location of point. See chapter 4 [Basic], page 19.

Point Pdl

The point pdl is used to hold several recent previous locations of point, just in case you want to move back to them. See section 10.3 [Point Pdl], page 41.

Possibilities List

A possibilities list is text in a Zmacs buffer that records several Lisp functions or buffer sections that a previous Zmacs command picked out for you to look at at your leisure. See section 26.7 [Possibilities Lists], page 164.

Prefix Character

A prefix character is a command whose sole function is to introduce a set of multi-character commands. Control-X (q.v.) is a prefix character. See section 2.2 [Multi-character Commands], page 11.

Prompt A prompt is text printed to ask the user for input. Printing a prompt is called *prompting*. Zmacs prompts always appear in the echo area (q.v.). One kind of prompting happens when the minibuffer is used to read an argument (see chapter 7 [Minibuffer], page 29); the echoing which happens when you pause in the middle of typing a multi-character command is also a kind of prompting (see section 1.3 [Echo Area], page 6).

Quitting Quitting is synonymous with aborting (q.v.), in Zmacs terminology. See section 32.1 [Quitting], page 205.

Quoting Quoting means depriving a character of its usual special significance. It is usually done with **Control-Q**. What constitutes special significance depends on the context and on convention. For example, an “ordinary” character as an EMACS command inserts itself; so in this context, a special character is any character that does not normally insert itself (such as **(RUBOUT)**, for example), and quoting it makes it insert itself as if it were not special. Not all contexts allow quoting. See chapter 4 [Basic Editing], page 19.

Read-only Buffer

A read-only buffer is one whose text you are not allowed to change. Normally Zmacs makes buffers read-only when they contain text which has a special significance to Zmacs; for example, Dired buffers. See chapter 19 [Buffers], page 99.

Reaping Reaping means deleting (q.v.) old versions of a file to recover disk space. See section 17.6 [Deleting Files], page 82.

Recursive Editing Level

A recursive editing level is a state in which part of the execution of a command involves asking the user to edit some text. This text may or may not be the same as the text to which the command was applied. The mode line indicates recursive editing levels with square brackets (‘[’ and ‘]’). See section 29.1 [Recursive Edit], page 179.

Redisplay

Redisplay is the process of correcting the image on the screen to correspond to changes that have been made in the text being edited. See chapter 1 [Screen], page 5.

Region The region is the text between point (q.v.) and the mark (q.v.), at times when the mark is active. At such times, the extent of the region is indicated on the screen by underlining. Many commands operate on the text of the region. See chapter 10 [Mark], page 39.

Registers Registers are named slots in which text or buffer positions can be saved for later use. See section 10.4 [Named Marks], page 42, for saving positions; See section 11.3.2 [Registers], page 51, for saving text in registers.

Replacement

See “global substitution”.

(RETURN) **(RETURN)** is the character that separates lines of text. It is also a Zmacs command to insert a **(RETURN)** into the text. It is also used to terminate most arguments read in the minibuffer (q.v.). See chapter 2 [Characters], page 11.

(RUBOUT) **(RUBOUT)** is a character used as a command to delete one character of text. See chapter 4 [Basic Editing], page 19.

S- **S-** in the name of a character is an abbreviation for **(SUPER)**, one of the modifier keys that can accompany any character. See chapter 2 [Characters], page 11.

S-expression

An s-expression is the basic syntactic unit of Lisp in its textual form: either a list, or

Lisp atom. Many Zmacs commands operate on s-expressions. See section 25.2 [Lists], page 136.

Saving Saving a buffer means copying its text into the file that was visited (q.v.) in that buffer. This is the way text in files actually gets changed by your Zmacs editing. See section 17.3 [Saving], page 79.

Scrolling Scrolling means shifting the text in the Zmacs window so as to see a different part of the buffer. See chapter 14 [Display], page 59.

Searching

Searching means moving point to the next occurrence of a specified string. See chapter 15 [Search], page 61.

Section A section is a portion of a Zmacs buffer used for one Lisp expression (or similar things in other languages). See section 26.1 [Sectionization], page 153.

Sectionization

Sectionization is the process or technique of dividing a Zmacs buffer into sections (q.v.).

Selecting Selecting a buffer means making it the current (q.v.) buffer. See chapter 19 [Buffers], page 99.

Self-documentation

Self-documentation is the feature of EMACS which can tell you what any command does, or give you a list of all commands related to a topic you specify. You ask for self-documentation with the **(HELP)** character. See chapter 9 [Help], page 37.

Sentences

Zmacs has commands for moving by or killing by sentences. See section 24.3 [Sentences], page 127.

Simultaneous Editing

Simultaneous editing means two users modifying the same file at once. Simultaneous editing if not detected can cause one user to lose his work. Zmacs detects all cases of simultaneous editing and warns the user to investigate them. See section 17.3.1 [Simultaneous Editing], page 80.

Sorting Sorting, in connection with Zmacs, means alphabetization. See section 29.4 [Sorting], page 181.

Source Compare

Source Compare is the facility in Zmacs that compares two files or buffers, either telling you what parts of them differ or merging the two. See section 17.7 [Source Compare], page 84.

String Substitution

See "global substitution".

Super

Super is the name of a modifier bit which a command character may have. It is present in a character if the character is typed with the **(SUPER)** key held down. Such

characters are given names that start with **Super-**. For example, **Super-Q** is typed by holding down **(SUPER)** and typing **Q**. See chapter 2 [Characters], page 11.

Syntax Table

The syntax table tells EMACS which characters are part of a word, which characters balance each other like parentheses, etc. See section 31.5 [Syntax], page 201.

Tag Table

Obsolete name for a buffer group (q.v.).

Text

Two meanings (see chapter 24 [Text], page 125):

Data consisting of a sequence of characters. The contents of a Zmacs buffer are always text in this sense.

Data consisting of written human language, as opposed to programs, or following the stylistic conventions of human language.

Top Level

Top level is the normal state of EMACS, in which you are editing the text of the file you have visited. You are at top level whenever you are not in a recursive editing level (q.v.) or the minibuffer (q.v.), and not in the middle of a command. You can get back to top level by aborting (q.v.).

Transposition

Transposing two units of text means putting each one into the place formerly occupied by the other. There are Zmacs commands to transpose two adjacent characters, words, s-expressions (q.v.) or lines (see section 16.2 [Transposition], page 72).

Typeout

Typeout is a message, printed by an EMACS command, which overwrites the area normally used for displaying the text being edited, but which does not become part of the text. Typeout is used for messages which might be too long to fit in the echo area (q.v.). See section 1.2 [Typeout], page 5.

Undoing

Undoing means making your previous editing go in reverse, bringing back the text that existed earlier in the editing session. See chapter 12 [Undo], page 53.

Variable

A Zmacs variable is a Lisp variable defined as part of Zmacs so that, by changing the value of the variable, you can adapt the behavior of Zmacs to your needs. All the Zmacs variables are listed in the variables index in this manual. See section 31.2 [Variables], page 192, for information on setting variables.

Visiting

Visiting a file means loading its contents into a buffer (q.v.) where they can be edited. See section 17.2 [Visiting], page 76.

Whitespace

Whitespace is any run of consecutive formatting characters (space, tab, newline, and backspace). The variable ***whitespace-chars*** tells Zmacs which characters to regard as whitespace.

Window

Zmacs uses windows to do its display on the screen. In particular, each Zmacs frame (q.v.) contains one or more Zmacs windows, each of which can display one buffer at

any time. See chapter 1 [Screen], page 5, for basic information on how Zmacs uses the screen. See chapter 20 [Windows], page 109, for commands to control the use of windows within the Zmacs frame.

Word Abbrev

Synonymous with "abbrev".

Word Search

Word search is searching for a sequence of words, considering the whitespace between them as insignificant. See section 15.3 [Word Search], page 64.

Yanking Yanking means reinserting text previously killed. It can be used to undo a mistaken kill, or for copying or moving text. See section 11.2 [Yanking], page 47.

Zmacs Zmacs is the program used for editing files on the Lisp Machine.

Zmacs Window

A Zmacs window is a window that displays the current contents of one Zmacs buffer. Most often a Zmacs frame contains one Zmacs window, but it can contain two, or possibly more. See chapter 20 [Windows], page 109.

Zmacs Frame

The Zmacs frame is a window, containing several panes, in which Zmacs does all of its screen display. The Zmacs window, the mode line, and the echo area are all panes of the Zmacs frame. See chapter 1 [Screen], page 5.

Ztop Ztop is a Zmacs facility that allows input and output for the Lisp listen loop and other stream-oriented programs to be done through a Zmacs buffer. This allows editing of both input and output, and keeps a permanent typescript of them, which you can save in a file. See chapter 27 [Ztop], page 171.

ZWEI ZWEI is the Lisp Machine editing system. Programs in this system include Zmacs (for editing files), ZMail (for editing incoming mail), and Converse (for editing interactive messages to and from other logged-in users).

Command Function Index

A

Add Global Word Abbrev	175
Add Mode Word Abbrev	176
Add Patch	167
Add Patch Buffer Changed Sections	167
Add Patch Changed Sections	167
Any Extended Command	201
Append Next Kill	49
Append To Buffer	50
Append To File	50, 88
Apropos	37
Arglist	151
Auto Fill Mode	115, 130

B

Back To Indentation	116
Backward	20
Backward Kill Sentence	45, 71, 127
Backward Kill Sexp	45, 137
Backward Kill Word	45, 71, 126
Backward List	137
Backward Paragraph	128
Backward Sentence	127
Backward Sexp	136
Backward Up List	137
Backward Word	126
Beginning Of Defun	138
Beginning Of Line	20
Break	183
Buffer Edit	103

C

Call Last Kbd Macro	196
Cancel Patch	168
Center Line	131
Change Default Font	122
Change File Properties	89
Change Font Char	122
Change Font Region	122
Change Font Word	122
Change One Font Region	122
Clean Directory	84

Clear	45
Comment Out Region	145
Compile And Exit	159
Compile And Load File	157
Compile Buffer	159
Compile Buffer Changed Sections	159
Compile Changed Sections	160
Compile File	157
Compile Region	158
Copy Binary File	88
Copy File	88
Copy Text File	88
Correct Word Spelling	73
Count Lines Page	130
Count Occurrences	68
Create Directory	89
Create Link	88

D

Define Character	200
Define Word Abbrevs	178
Delete ()	150
Delete Blank Lines	22, 46
Delete File	83
Delete Forward	45
Delete Horizontal Space	46, 116
Delete Indentation	46, 116
Delete Matching Lines	68
Delete Non-Matching Lines	68
Describe	37
Describe Flavor	164
Describe Variable	193
Describe Variable At Point	151
Dired	91
Disassemble	163
Discard Undo Information	55
Display Directory	82
Display Font	121
Dissociated Press	185
Down Comment Line	144
Down List	137
Down Real Line	20

E

Edit Buffer Changed Sections	155
Edit Callers	163
Edit Changed Sections	156
Edit Combined Methods	164
Edit Definition	154
Edit File Warnings	161
Edit Flavor Components	164
Edit Flavor Dependents	164
Edit Flavor Direct Dependents	164
Edit Flavor Methods	164
Edit Methods	164
Edit Next Warning	160
Edit Object Users	163
Edit Options	192
Edit Previous Warning	160
Edit System Warnings	161
Edit Tab Stops	117, 125
Edit Warnings	161
Edit Word Abbrevs	178
Electric Font Lock Mode	122
Electric Shift Lock Mode	147
End Kbd Macro	196
End Of Defun	138
End Of Line	20
Evaluate And Exit	159
Evaluate And Replace Into Buffer	183
Evaluate Buffer	159
Evaluate Buffer Changed Sections	159
Evaluate Changed Sections	160
Evaluate Into Buffer	182
Evaluate Mini Buffer	182
Evaluate Region	158
Evaluate Region Hack	159
Evaluate Region Verbose	159
Exchange Characters	20, 72
Exchange Lines	72
Exchange Regions	72
Exchange Sexps	72, 137
Exchange Words	72, 126
Execute Command Into Buffer	6
Expunge Directory	83
Extended Command	35

F

Fast Where Am I	9
Fill Long Comment	132, 145
Fill Paragraph	131
Fill Region	131
Find Alternate File	78
Find File	77
Find File Background	78
Find File No Sectionize	78, 156
Find System Files	78
Find Unbalanced Parentheses	162
Finish Patch	168
Finish Patch Unreleased	168
Finish Ztop Evaluation	173
Forward	20
Forward List	137
Forward Paragraph	128
Forward Sentence	127
Forward Sexp	136
Forward Up List	137
Forward Word	126
Frob Do	149
Frob Lisp Conditional	148

G

Get Register	51
Go To AC Field	183
Go To Address Field	183
Go To Next Label	183
Go To Next Possibility	164
Go To Previous Label	183
Goto Beginning	20
Goto Character	20
Goto End	20
Grow List Backward	150
Grow List Forward	150
Grow Window	111

I

Incremental Search	61
Indent Differently	139
Indent For Comment	143
Indent For Lisp	135, 139
Indent Nested	118

Indent New Comment Line	144
Indent New Line	115, 139
Indent Region	116, 140
Indent Relative	118
Indent Rigidly	116
Indent Sexp	140
Indent Under	118
Insert Crs	19
Insert File	88
Insert File No Fonts	88
Insert Tab	115
Insert Word Abbrevs	178
Install Macro	197
Install Mouse Macro	197

J

Jump To Register Position	42
Just One Space	46

K

Kbd Macro Query	198
Kill All Word Abbrevs	176
Kill Buffer	102
Kill Comment	144
Kill Global Word Abbrev	176
Kill Line	45
Kill Local Variable	194
Kill Mode Word Abbrev	176
Kill Or Save Buffers	103
Kill Region	45, 47
Kill Sentence	45, 127
Kill Sexp	45, 137
Kill Some Buffers	102
Kill Terminated Word	183
Kill Word	45, 126

L

Lisp Match Search	65
List Buffer Changed Sections	155
List Buffers	100
List Callers	163
List Changed Sections	156
List Combined Methods	164
List Files	82

List Flavor Components	164
List Flavor Dependents	164
List Flavor Direct Dependents	164
List Flavor Methods	164
List Fonts	121
List Local Variables	194
List Matching Lines	68
List Matching Symbols	162
List Methods	164
List Object Users	163
List Registers	52
List Sections	155
List Some Word Abbrevs	177
List Tag Tables	105
List Variables	194
List Word Abbrevs	177
Load File	157
Long Documentation	151
Lowercase Lisp Code In Region	148
Lowercase Region	119
Lowercase Word	72, 119

M

Macro Expand Expression	163
Macro Expand Expression All	163
Mail	179
Make ()	149
Make Local Variable	194
Make Room	22
Make Word Abbrev	176
Mark Beginning	41
Mark Defun	41, 138
Mark End	41
Mark Page	41, 129
Mark Paragraph	41, 128
Mark Sexp	41, 137
Mark Whole	41
Mark Word	41, 127
Microcompile Region	159
Midas Mode	183
Modified Two Windows	111
Move Over)	149
Move To Screen Edge	60
Multiple Edit Callers	163

Multiple Edit Object Users	163	Rename File	88
Multiple List Callers	163	Reparse Attribute List	188
Multiple List Object Users	163	Repeat Mini Buffer Command	33
Multiple Query Replace	67	Replace String	65
Multiple Query Replace From Buffer	67	Reposition Window	60
N		Require Activation Mode	173
Name Last Kbd Macro	197	Resume Patch	168
New Window	59	Reverse Following List	150
Next File	107	Reverse Incremental Search	61
Next Page	129	Reverse String Search	63
Next Screen	59	Revert Buffer	81
Not Modified	79	Rubout	19, 45, 71
O		S	
One Window	110	Save All Files	79
Other Window	110	Save File	79
P		Save Position In Register	42
Pop Mini Buffer History	33	Save Region	48
Prepend To File	50, 88	Scroll Other Window	110
Previous Page	129	Sectionize Buffer	156
Previous Screen	59	Select All Buffers As Tag Table	105
Print File	88	Select Buffer	99
Print Modifications	53	Select Default Previous Buffer	100
Put Register	51	Select Last Ztop Buffer	171
Q		Select Previous Buffer	100
Query Exchange	67	Select Some Buffers As Tag Table	105
Query Replace	66	Select System As Tag Table	105
Query Replace Last Kill	67	Select Tag Table	105
Query Replace Let Binding	149	Set Backspace	189
Quick Arglist	151	Set Base	189
Quick Documentation	151	Set Comment Column	146
Quick Redo	53	Set Common Lisp	189
Quick Undo	53	Set Fill Column	131
Quoted Insert	19	Set Fill Prefix	132
R		Set Fonts	120, 189
Read Word Abbrev File	178	Set Lowercase	148, 189
Reap File	84	Set Nofill	189
Recenter Window	20	Set Package	189
Release Patch	168	Set Patch File	189
Rename Buffer	102	Set Pop Mark	40
		Set Readtable	189
		Set Tab Width	117, 189
		Set Variable	193
		Set Visited File Name	80

Set Vsp	189
Show List Start	143
Sort Decreasing Creation Date	96
Sort Decreasing File Name	96
Sort Decreasing Reference Date	96
Sort Decreasing Size	96
Sort Increasing Creation Date	96
Sort Increasing File Name	96
Sort Increasing Reference Date	96
Sort Increasing Size	96
Sort Lines	181
Sort Pages	181
Sort Paragraphs	181
Sort Via Kbd Macros	181
Source Compare	85
Source Compare Changes	85
Source Compare Merge	86
Split Line	116
Stack List Vertically	140
Start Kbd Macro	196
Start Patch	167
Start Private Patch	169
String Search	63
Swap Point and Mark	40

T

Tab Hacking Rubout	135
Tab To Tab Stop	117, 125
Tags Add Patch Changed Sections	167
Tags Compile Buffer Changed Sections	160
Tags Compile Changed Sections	107
Tags Edit Changed Sections	107, 156
Tags Evaluate Buffer Changed Sections	160
Tags Evaluate Changed Sections	107
Tags List Changed Sections	107, 156
Tags Multiple Query Replace	106
Tags Multiple Query Replace From Buffer	106
Tags Query Replace	106
Tags Search	106
Tags Search List Sections	107
Text Formatter Change Font Region	133
Text Formatter Change Font Word	133
Text Mode	125
This Indentation	116
Toggle Read Only	102

Trace	162
Two Windows	110
Two Windows Showing Region	111

U

Undefine Character	200
Unexpand Last Word	177
Universal Argument	27
Up Comment Line	144
Up Real Line	20
Uppcase Digit	72
Update Attribute List	190
Uppercase Initial	72, 119
Uppercase Lisp Code In Region	148
Uppercase Region	119
Uppercase Word	72, 119

V

Variable Apropos	194
Various Quantities	57
View File	87
View Kbd Macro	196
View Two Windows	110
Visit Tag Table	105

W

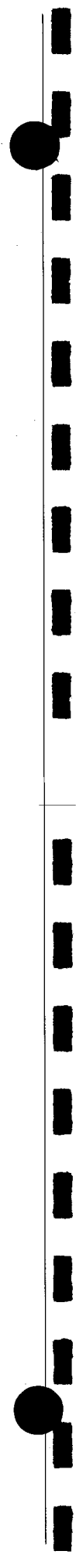
Where Am I	9
Where Is Symbol	162
Word Abbrev Mode	175
Write File	80
Write Region To File	88
Write Word Abbrev File	178

Y

Yank	48
Yank Default String	29
Yank Pop	49
Yank Previous Input	29, 76
Yank Search String	30


Z

Ztop Abort	173
Ztop Abort All	173
Ztop Mode	171
Ztop Yank Input History	173

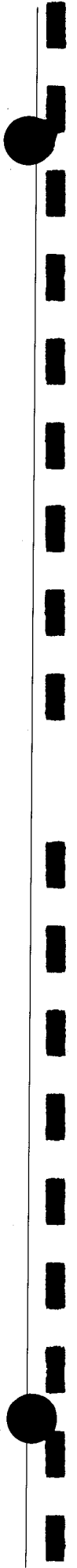


Command Character Index

!		C-MF	136
!	66	C-MH	41, 138
.		C-MJ	122
.	66	C-MK	45, 137
A		C-ML	100
Abort	173, 205	C-MM	116
Altmode	31, 66	C-MN	137, 183
		C-MO	116
		C-MP	137, 183
		C-MQ	140
		C-MR	60
		C-MRubout	45, 137
		C-MShift-B	150
		C-MShift-C	159
		C-MShift-F	150
		C-MShift-K	150
		C-MT	72, 137
		C-MTab	139
		C-MU	137
		C-MV	110
		C-MW	49
		C-MX	201
		C-MY	29, 76, 173
		C-MZ	159
		C-M $\text{\textcircled{C}}$	41, 137
		C-N	20
		C-O	22
		C-P	20
		C-Q	19
		C-R	61, 66
		C-Rubout	135
		C-S	61
		C-Shift-A	151
		C-Shift-C	158
		C-Shift-D	151
		C-Shift-E	158, 159
		C-Shift-F	101
		C-Shift-J	122
		C-Shift-P	164
		C-Shift-R	53
		C-Shift-S	30, 65
B			
Break	183		
C			
C-%	65		
C-)	143		
C-;	143		
C=	9		
C-A	20		
C-Abort	205		
C-B	20		
C-Backslash	46		
C-D	45		
C-E	20		
C-F	20		
C-G	77, 205		
C-J	122		
C-K	45		
C-L	20, 59, 66		
C-M'	20		
C-Mft	149		
C-M&	148		
C-M(137		
C-M)	137		
C-M;	144		
C-M[138		
C-M]	138		
C-MA	138, 183		
C-MB	136		
C-MBackslash	116, 140		
C-MD	137, 183		
C-ME	138, 183		

C-Shift-U	53	C-X C-T	72
C-Shift-V	151	C-X C-U	119
C-Shift-W	160	C-X C-V	78
C-Shift-X	57	C-X C-W	80
C-Shift-Y	29, 76	C-X C-X	40
C-Space	40	C-X D	91
C-T	20, 72	C-X E	196
C-Tab	139	C-X F	131
C-U	27, 77	C-X G	51
C-U C-Space	41	C-X H	41
C-U C- 	128	C-X J	42
C-V	59	C-X K	102
C-W	45, 47, 66, 128	C-X L	130
C-X	11	C-X M	179
C-X '	133	C-X O	110
C-X (196	C-X Q	198
C-X)	196	C-X Rubout	45, 71, 127
C-X +	175	C-X S	42
C-X .	132	C-X T	72
C-X 1	110	C-X Tab	116
C-X 2	110	C-X U	177
C-X 3	110	C-X X	51
C-X 4	111	C-X ^	111
C-X 8	111	C-Y	48
C-X ;	146	C->	41
C-X =	9	C-<	41
C-X [129	Clear-input	45
C-X]	129	Clear-screen	59
C-X A	50	Comma	66
C-X Altmode	33	E	
C-X B	99	End	173, 179
C-X C;	145	H	
C-X C-A	176	Help	37
C-X C-B	100	L	
C-X C-D	82	Linefeed	113, 115, 139
C-X C-F	77	M	
C-X C-J	122	M'	133
C-X C-L	119	M-ft	73
C-X C-M-L	100	M-%	66
C-X C-O	22, 46	M(149
C-X C-P	41, 129		
C-X C-Q	102		
C-X C-S	79		

M)	149	M-Rubout	45, 71, 126
M*	72	M-S	131
M- M-C	72	M-Shift-E	159
M- M-L	72	M-Shift-J	122
M- M-U	72	M-Shift-W	160
M.	154	M-Shift-Y	33
M;	143	M-T	72, 126
M[128, 183	M-Tab	115
M]	128, 183	M-U	119
M-A	127	M-V	59
M-Aabort	173	M-W	48
M-Altmode	182	M-X	35, 201
M-B	126	M-Y	49
M-Backslash	46, 116	M-Z	159
M-C	119	M~	79
M-D	45, 126	M->	20
M-E	127	M^	46, 116
M-F	126	M-<	20
M-G	131	M@	41, 127
M-H	41, 128	Macro C	197
M-I	117	Macro M	197
M-J	122	R	
M-K	45, 127	Return	19
M-L	119	Rubout	19, 45, 66, 71, 113, 135, 192
M-Linefeed	144	S	
M-M	116	Space	66, 73
M-N	144	T	
M-O	116	Tab	113, 115, 125, 135, 139
M-P	144		
M-Q	131		
M-R	60		



LISP Function Index

D

direc 15

E

ed 15

M

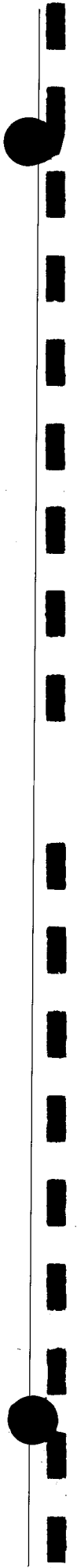
mail 16

Z

zwei:edit-functions 15

zwei:load-directory-into-zmacs 16

zwei:load-file-into-zmacs 16



Variable Index

*	
alphanumeric-case-affects-search	63, 66
auto-directory-display	82
auto-fill-activation-characters	131
auto-push-point-notification	42
auto-push-point-option	41
blanks	46
case-replace-p	66
center-fraction	60
check-unbalanced-parentheses-when-saving	162
comment-begin	146
comment-column	146
comment-end	146
comment-round-function	146
comment-start	146
cond-clause-superiors	142
default-base	187
default-major-mode	114
default-package	187
default-readtable	187
default-zmacs-bug-template	180
default-zmacs-mail-template	180
discard-undo-info-on-saving	55
electric-shift-lock-xors	147
file-versions-kept	84
fill-column	131
fill-extra-space-list	131
fill-prefix	132
find-file-early-select	78
find-file-not-found-is-an-error	78
flash-matching-paren	143
flash-matching-paren-max-lines	143
history-menu-length	50
history-rotate-if-numeric-arg	50
history-yank-wraparound	49
indent-not-function-superiors	142
indent-with-tabs	115
initial-minor-modes	191
lisp-defun-indentation	141
lisp-indent-lone-function-offset	140
lisp-indent-offset	140
lisp-indent-offset-alist	141
major-mode-translations	114
max-reset-fraction	60
min-reset-fraction	60
next-screen-context-lines	60
one-window-default	110
page-delimiter-list	130
paragraph-delimiter-list	129
point-pdl-max	42
region-marking-mode	41
region-right-marking-mode	41
space-indent-flag	115
tab-blinker-flag	5
temp-file-type-list	84
text-justifier-escape-list	129
transfer-minor-modes	191
undo-save-small-changes	54
unsticky-minor-modes	191
whitespace-chars	46
F	
fs:*file-type-mode-index*	113



Concept Index

- !
- ! 19
- A**
- abbrevs 175
- accumulator 183
- activation (Ztop) 172
- address 183
- againformation 185
- assembler code 183
- Atom Word mode 127, 192
- attribute line 187
- Auto Fill mode 130, 144, 191
- B**
- blank lines 22, 144
- Bolio 133
- BoTeX 133
- buffer group 104
- buffer selection history 100
- buffers 99
- buggestion 185
- C**
- C 11
- case conversion 72, 119, 146
- centering 131
- changed sections 155
- character set 11
- command function 12, 199
- command name 199
- command table 12
- comments 143, 203
- compilation 157
- completion 30
- comtab 12, 201
- confirmation 119
- connected 12
- continuation line 19
- Control 11
- Control-Meta 136
- creating files 78
- current buffer 99
- current font 122
- cursor 5, 19
- customization 12, 140, 191
- D**
- debugger 16
- default argument 29, 76
- default file names 89
- defuns 138
- deletion 19, 45
- deletion (of files) 83, 91
- directory hierarchy 93
- directory listing 82
- Dired 91
- don't reap bit 84
- dormant (Ztop) 172
- drastic changes 81
- E**
- echo area 6
- Electric Font Lock mode 122, 191
- Electric Shift Lock mode 147, 191
- Emacs mode 192
- evaluation 157
- exchanging 126
- expunging files 83
- extended command 35
- extended search 68
- F**
- file attributes 93, 187
- file dates 80
- file directory 81
- file names 75
- file properties 93
- files 21, 75, 77
- fill prefix 132
- filling 130
- flavors 164
- Font attribute 120
- fonts 120, 133

- formatting 139
 functions 35
- G**
- global substitution 65
 graphic characters 19
 grinding 139
- H**
- H- 11
 hardcopy 180
 help 37
 Hyper 11
- I**
- ignoriginal 185
 indentation 115, 139, 143
 insertion 19
 interpretation (of Lisp) 157
- J**
- justification 131
- K**
- keyboard macros 195
 kill history 47
 killing 23, 45
- L**
- label (of a window) 109
 labels 183
 Lisp 135
 Lisp mode 135
 Lisp Pattern Search 65
 lists 136
 local variables 194
- M**
- M- 11
 M-X commands 35
 Macsyma mode 184
 mail 179
 major modes 113, 146
 mark 39
- Markov chain 185
 matching parentheses 143
 message 179
 Meta 11, 126
 microcompilation 159
 MIDAS 183
 minibuffer 29, 35
 minor modes 191
 Mode attribute 113
 mode line 7, 191
 modified (buffer) 77
 mouse . . . 8, 23, 30, 32, 33, 82, 121, 151, 155, 163, 164
 moving text 47
- N**
- named marks 42
 newline 19
 Nofill attribute 125
 nonincremental search 63
 numeric arguments 27
- O**
- options 192
 outragedy 185
 Overwrite mode 192
- P**
- pages 129
 paragraphs 128
 parentheses 143
 patches 165
 point 5, 19
 point pdl 41
 possibilities list 164
 prefix characters 11
 prompting 7
 properbose 185
- Q**
- Query Replace 66
 quitting 205
 quoting 19
- R**
- R (text formatter) 133

read-only buffer	102
reaping old versions	83
recursive edit	179
region	39, 119
registers	42, 51
replacement	65
Return Indents mode	192

S

S	11
s-expressions	136
saving	77
screen	5
scrolling	24, 59
search string history	64
searching	61
sectionization	153
selected buffer	99
self-documentation	37
sentences	127
simultaneous editing	80
sorting	181
source compare	84
spelling	73
string substitution	65
Super	11
syntax table	127, 201

T

Tab Width attribute	117
tag table	104
techniquitous	185
text	125
text formatters	133

Text mode	125
top level	7
transposition	72, 137
typeout	5
typos	71, 72

U

underlining	133
undo	53, 119
unreleased patches	168

V

variables	12, 192
version numbers (of a system)	166
viewing	87
visiting	77
visiting files	76
Vsp attribute (of a file)	121

W

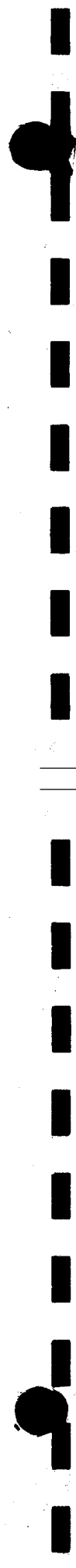
windows	109
Word Abbrev mode	192
word abbrevs	175
word search	64
words	72, 119, 126

Y

yanking	23, 47
---------	--------

Z

Zmacs frame	5
Ztop	171
ZWEI	13



MINCE

Distributed by LMI 6033 W. Century Blvd. Los Angeles CA 90045
USA

Disclaimer

The seller of the software described in this manual hereby disclaims any and all guarantees and warranties, both express and implied. No liability of any form shall be assumed by the seller, nor shall direct, consequential, or other damages be assumed by the seller. Any user of this software uses it at his or her own risk.

This product is sold on an "as is" basis; no warranty of merchantability is claimed or implied.

Due to the ill-defined nature of "fitness for purpose" or similar types of warranties for this type of product, no fitness for any purpose whatsoever is claimed or implied.

The seller reserves the right to make changes, additions, and improvements to the software at any time; no guarantee is made that future versions of the software will be compatible with any other version.

The parts of this disclaimer are severable and fault found in any one part does not invalidate any other parts.

Copyright © 1981 by Mark of the Unicorn Inc.

Welcome

We strongly suggest that you begin by reading Section 1, the Installation Guide. This document will help you get started with as little pain as possible.

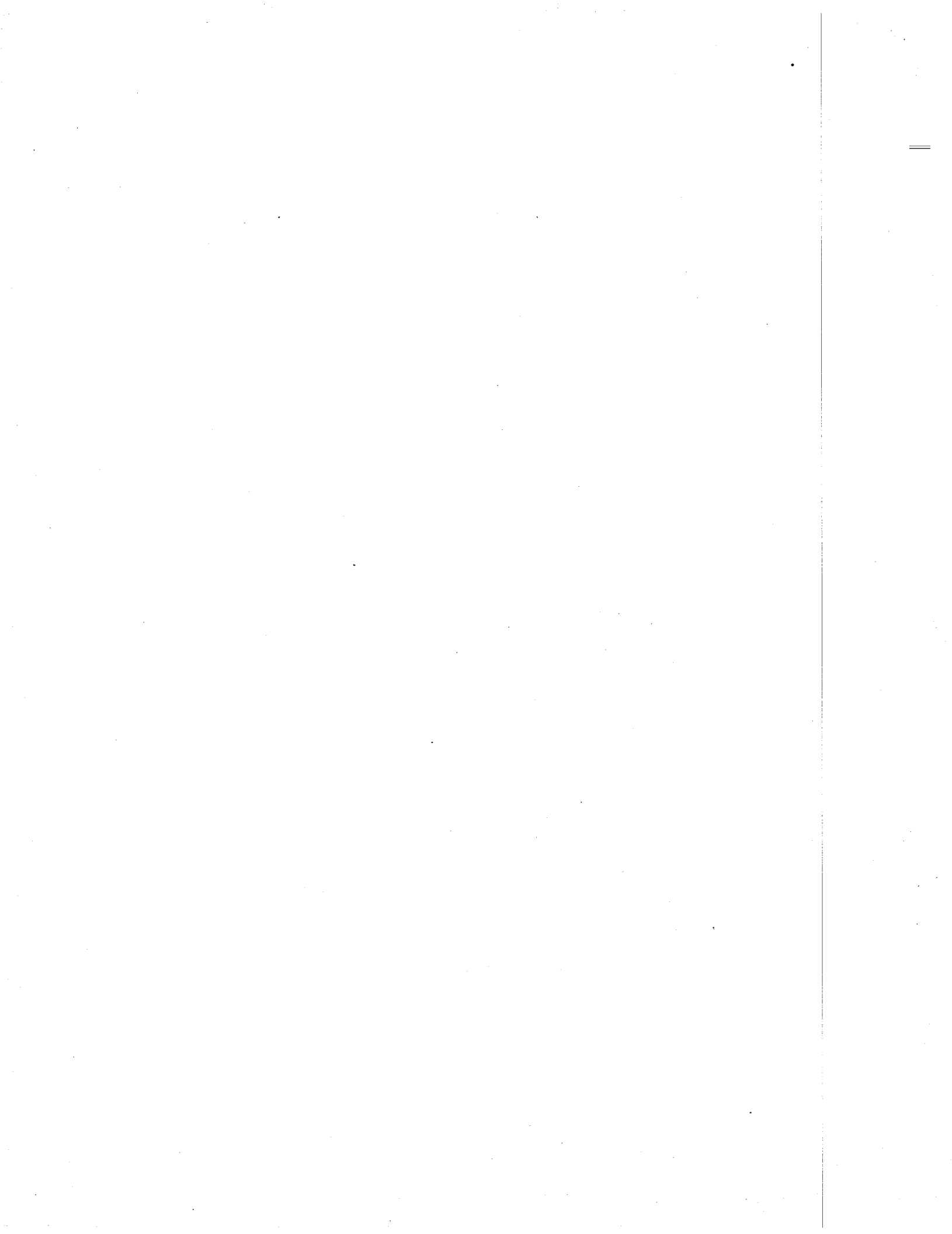
However, if you are like us, you fiddle first and read the instructions only as a last resort. In that case it is now time to turn to Section 1, the Installation Guide. It will tell you how to really run Config and what to do after you botched it.

On the other hand, if you don't really want instructions, then turn to the Installation Guide (Section 1, in case you didn't know) anyway, just as a favor to us. Sorry, you can't NOT read the Installation Guide!

If you are a new user, you should begin by reading/doing either the Mince Lessons or the Programmer's Introduction. At a later time you may wish to read the User's Guide; it explains the terminology and concepts used in and around Mince. While you are learning and first using Mince, the Command Summary, which lists each command and its action, will be useful beside your terminal.

Enjoy,

Mark of the Unicorn
P.O. Box 423
Arlington, Massachusetts 02174



SINGLE CPU LICENSE AGREEMENT

Mark of the Unicorn Inc., P.O. Box 423, Arlington, Massachusetts (referred to as "Seller") and the party who executes this license (referred to as "Buyer") agree to the grant and acceptance of this License under the following terms and conditions:

1. The use of the words "supplied software" shall mean any and all programs or parts of programs in either source or object code form as well as any and all documentation, supplied in either hardcopy or machine-readable form, which are sold by the Seller under the product name Mince.
2. The Seller grants a nonexclusive license to the Buyer to use the supplied software.
3. This license includes the right to make a reasonable number of copies of the supplied software for backup purposes and ease of use. The Seller's copyright shall be extended to cover these copies and placed on all of these copies.
4. This license does not include the right to resell any portion of the supplied software, either alone or as part of another product, with the following exceptions:
 - 4.1. The Buyer has the right to sell any products developed with the aid of the supplied software in which the supplied software played only an ancillary role.
 - 4.2. The Buyer has the right to sell any products which are derived from the supplied software and the sale of which has been approved by the Seller.
5. The Seller's standard disclaimer shall apply to the supplied software.
6. The Buyer agrees to use the supplied software only on a single computer system. A description of this computer system shall be included as a part of this agreement. This description shall be in enough detail that it refers to a unique computer system. An example of such detail is the manufacturer's name, the model number, and the serial number.
7. The single computer system on which the supplied software may be used can be changed by ten (10) days' written notice to the

Seller. This notice shall include a description of the new computer system as in Section 6.

8. Breaking the seal on the envelope containing the machine-readable portions of the supplied software implies the acceptance of this agreement by the Buyer. Such acceptance does not invalidate the need for the Buyer to complete and return a copy of this agreement to the Seller.

9. The parts of this agreement are severable and fault found with any one part does not invalidate the others.

----- Buyer -----

Signed _____

Name _____

Title _____

Company _____

Address _____

Date _____

Description of the Single Computer System:

----- Seller -----

Mark of the Unicorn Inc.
P.O. Box 423
Arlington, Massachusetts 02174

SINGLE CPU LICENSE AGREEMENT

Mark of the Unicorn Inc., P.O. Box 423, Arlington, Massachusetts (referred to as "Seller") and the party who executes this license (referred to as "Buyer") agree to the grant and acceptance of this License under the following terms and conditions:

1. The use of the words "supplied software" shall mean any and all programs or parts of programs in either source or object code form as well as any and all documentation, supplied in either hardcopy or machine-readable form, which are sold by the Seller under the product name Mince.
2. The Seller grants a nonexclusive license to the Buyer to use the supplied software.
3. This license includes the right to make a reasonable number of copies of the supplied software for backup purposes and ease of use. The Seller's copyright shall be extended to cover these copies and placed on all of these copies.
4. This license does not include the right to resell any portion of the supplied software, either alone or as part of another product, with the following exceptions:
 - 4.1. The Buyer has the right to sell any products developed with the aid of the supplied software in which the supplied software played only an ancillary role.
 - 4.2. The Buyer has the right to sell any products which are derived from the supplied software and the sale of which has been approved by the Seller.
5. The Seller's standard disclaimer shall apply to the supplied software.
6. The Buyer agrees to use the supplied software only on a single computer system. A description of this computer system shall be included as a part of this agreement. This description shall be in enough detail that it refers to a unique computer system. An example of such detail is the manufacturer's name, the model number, and the serial number.
7. The single computer system on which the supplied software may be used can be changed by ten (10) days' written notice to the

Seller. This notice shall include a description of the new computer system as in Section 6.

8. Breaking the seal on the envelope containing the machine-readable portions of the supplied software implies the acceptance of this agreement by the Buyer. Such acceptance does not invalidate the need for the Buyer to complete and return a copy of this agreement to the Seller.

9. The parts of this agreement are severable and fault found with any one part does not invalidate the others.

----- Buyer -----

Signed _____
Name _____
Title _____
Company _____
Address _____

Date _____

Description of the Single Computer System:

----- Seller -----

Mark of the Unicorn Inc.
P.O. Box 423
Arlington, Massachusetts 02174

Disclaimer

The seller of the software described in this manual hereby disclaims any and all guarantees and warranties, both express and implied. No liability of any form shall be assumed by the seller, nor shall direct, consequential, or other damages be assumed by the seller. Any user of this software uses it at his or her own risk.

This product is sold on an "as is" basis; no warranty of merchantability is claimed or implied.

Due to the ill-defined nature of "fitness for purpose" or similar types of warranties for this type of product, no fitness for any purpose whatsoever is claimed or implied.

The physical medium upon which the software is supplied is guaranteed for one year against any physical defect. If it should fail, return it to Mark of the Unicorn, and a new physical medium with a copy of the purchased software shall be sent.

The seller reserves the right to make changes, additions, and improvements to the software at any time; no guarantee is made that future versions of the software will be compatible with any other version.

The parts of this disclaimer are severable and fault found in any one part does not invalidate any other parts.

Copyright (c) 1981 by Mark of the Unicorn Inc.

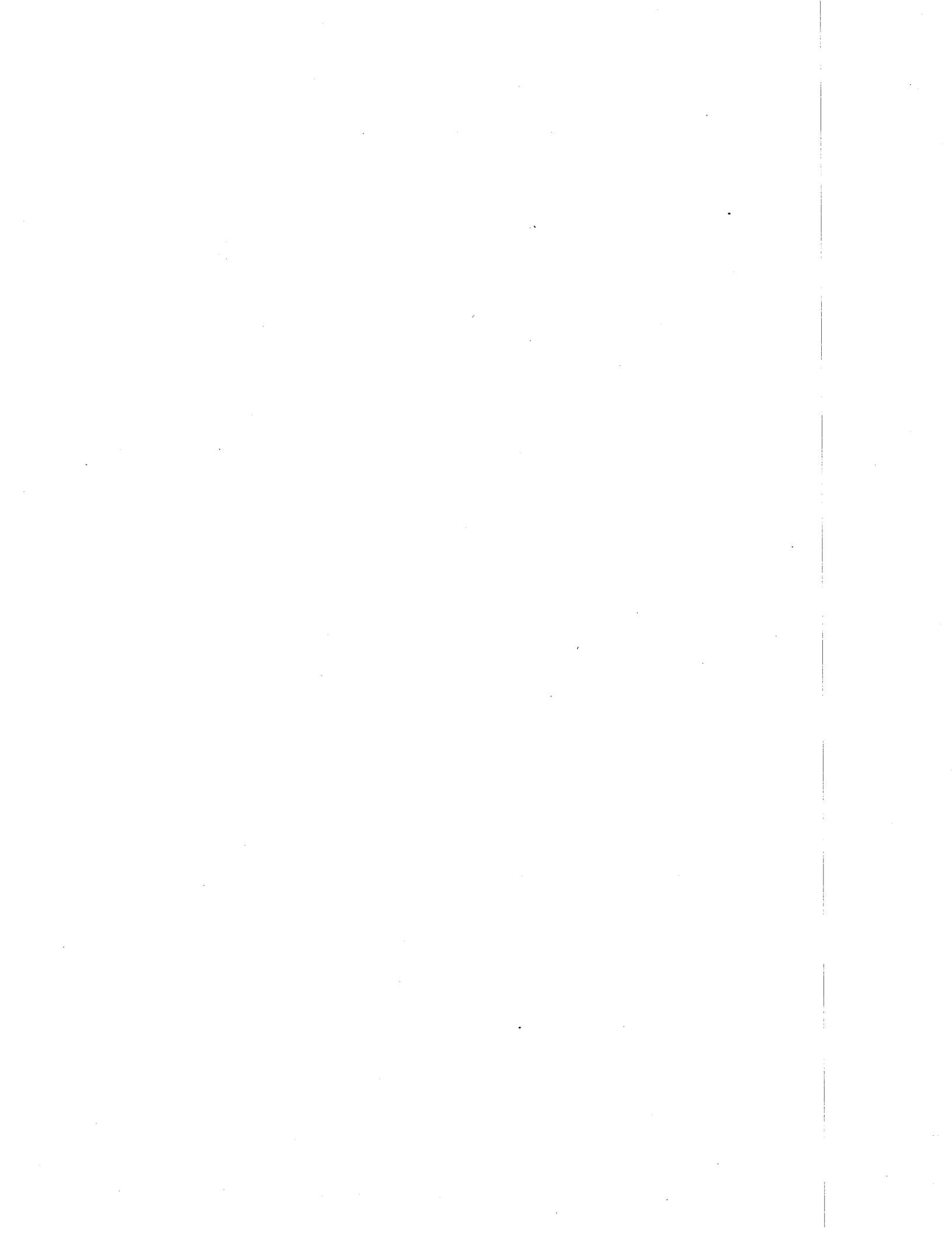


Table of Contents

Welcome message

1. Installation Guide

Tells you how to run Config and what to do if Mince doesn't come up.

2. Mince Lessons

Provides an introduction to Mince for people who are not familiar with computers or programming. Parts of it are simply read while other parts assume that you have a Mince available and talk you through many of the commands. (Part of this document is also supplied on line in the files LESSON4.DOC, LESSON6.DOC, AND LESSON8.DOC.)

3. Programmer's Introduction to Mince

Provides an introduction to Mince for people who are already familiar with at least one text editor. It is intended to be read into Mince and read while following the contained directions.

(This document is also supplied on line in the file PRGINTRO.DOC.)

4. Mince User's Guide

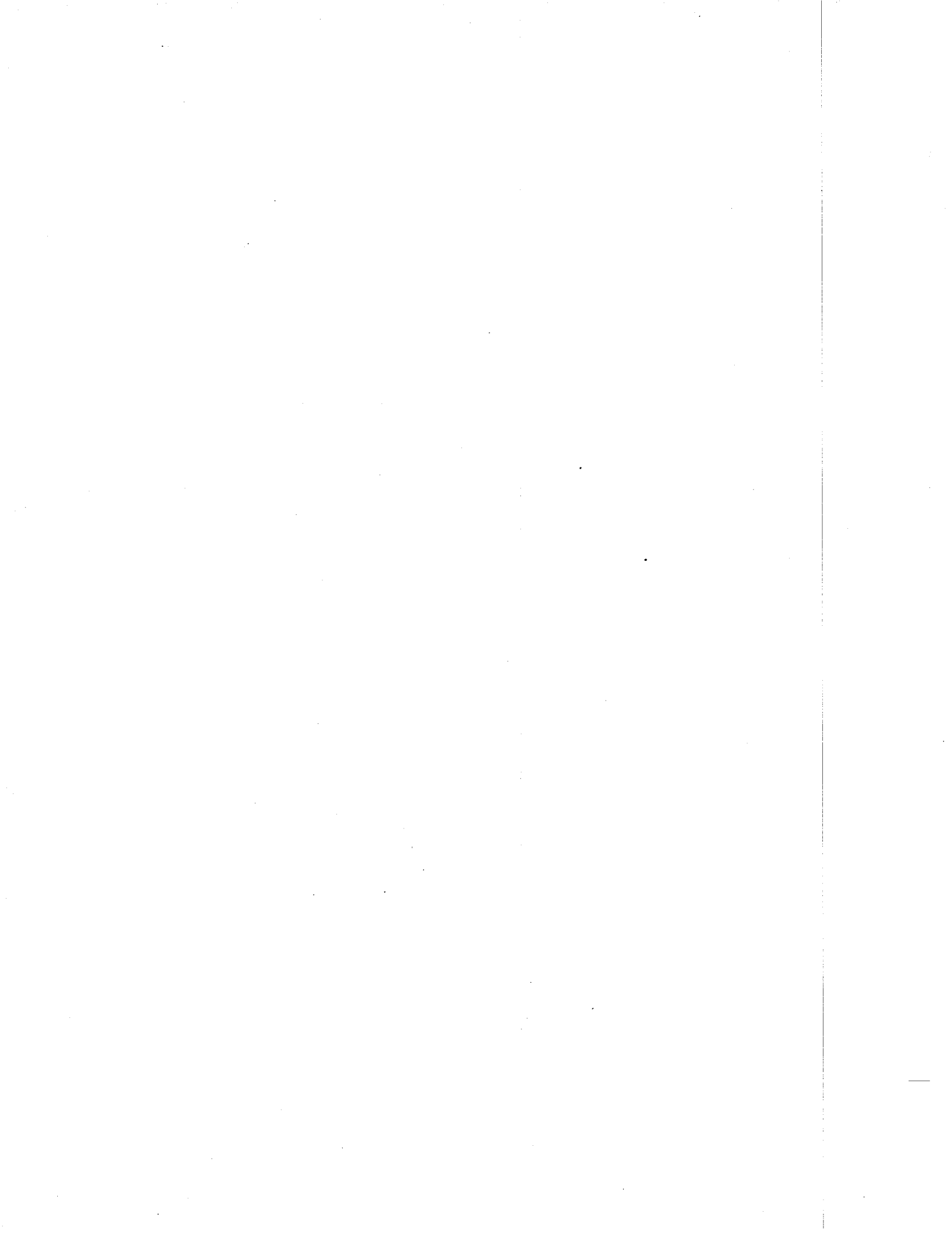
Explains the basic principles you should understand in order to use Mince.

5. Complete Command List

Lists each command and exactly what it does.

6. Command Summary

Lists each command.



Installation Guide for Mince

Mince is a program that needs to know a variety of things about your terminal and system. There are also some things you have to do before you can start using Mince. The process of telling Mince about your system and performing these other tasks is called installation. This guide will help you go through the installation process smoothly and with as few problems as possible; we suggest you read the whole thing once before starting.

As always, but especially during the installation process, if you need any help, feel free to call or write. Our address is:

Mark of the Unicorn
P.O. Box 423
Arlington, Massachusetts 02174
617-489-1387

We will provide all of the help that we can in order that you successfully bring up Mince. If you are calling us, it would help tremendously if you track down the hardware manuals for your system and terminal and have them at hand; we may need to find out something from them in order to help you.

First Step: Backing Up Disks

The first step in installing Mince is to make a backup copy of the original distribution disk (the one we sent you), then put the original away (preferably in another room, if not another building) for safekeeping.

A backup copy is simply an exact, "image" copy of an "original" disk. There are two reasons for making backup copies. First, if you are using a disk heavily, a backup should be made periodically in case the main disk gets damaged physically or "bashed" (the information on it gets so scrambled that it is unusable). Second, if you happen to make a mistake when doing something, a backup copy allows you to recover from that mistake quickly. Don't be afraid to have lots of copies of your disks; disks aren't expensive but your data is, and there is safety in copies. (Remember, a computer is, among other things, a device for making more mistakes faster than ever before possible!)

Both reasons for making backups apply to Mince. If you make a mistake in the installation process, you can retrieve the original disk, make another copy, and start over. Second, after Mince is installed and in use, you will want to be able to recover in case a disk gets damaged or bashed.

Thus, to repeat: the first step is to make a copy of the original distribution disk. Then, use the copy for all of the following steps, and put the original disk away for safekeeping.

(The following discussion is only interesting if you are interested in a fine point or two.)

The distribution disk is set up with a swap file and a Mince .COM file. By making copies of this disk, all other disks will have these as the first two files. It helps performance tremendously to make these the first two files on every disk you put them on.

(End of fine point.)

The Swap File

Mince uses something called a swap file. This file is a place to put parts of the documents that you are not currently editing (Mince "swaps" parts of your document to and from this file.) as well as a place for it to remember the specific details of your system. This file is called "MINCE.SWP" and there must always be a copy of it around that Mince can find.

The size of this file is important: it limits the total amount of text you can be editing at one time. A workable minimum is 24 pages or so, and the maximum size is 248 pages. Roughly, a 64 page swap file lets you edit files totalling up to 64K in size, but in fact the swap space is not completely utilized -- there are gaps -- and editing tends to increase the number of these gaps. For example, a 20K file will take up 20 pages when just read in, but might grow to 35 pages during a long and heavy editing session, even if you don't enter any new text. Thus, it is wise to leave yourself more space than you think you will need. We use a 64 page swap file and it seems to work quite well.

There is a swap file on the distribution disk that we sent to you. There will thus also be such a file on the copy of the distribution disk that you are currently using. You will not have to create another one unless you want to copy the Mince program onto other disks.

Second Step: The Configuration Program

Next, run the configuration program, "CONFIG". Config will lead you by the hand through the process of answering numerous questions. If you are in doubt about what to do, you should execute all of the steps from 1 to 8 in order.

Note that Config is a program that sometimes seems to be in a hurry. Many times you merely type the first letter of an answer (e.g., "y" or "n") and Config will fill in the rest of the word for you and go on from there. Other times, when you are told what the default answer is, you merely have to type a carriage return and that default answer will be used automatically.

The copy of the swap file that comes on the distribution disk (and is consequently on your backup) is set up for a Heath H19 terminal. If you have a different terminal, you must be sure to use step 2 of the configuration program so that you can tell Mince about it.

Third Step: What To Do If Something Goes Wrong

One possible problem is that the configuration program cannot write out the new version of the swap file. This will happen if you make the swap file larger than the space available on the disk. If the disk has other files on it, then you should find a blank disk and try again.

If the configuration program's terminal selection menu does not mention your terminal, you have an adventure ahead of you. You get to select the "not listed" option (option 1). Config proceeds to ask you lots of detailed questions about things that you might never have heard of. If you know the answers, fantastic! If not, try to look the answers up in your terminal manual, find a local expert, or call us. If you must call, it would help a great deal if you have the manufacturer's manual for your terminal at hand. If we had the manuals here, we would already have entered the terminal's characteristics!

If you use a video board as a terminal, you may or may not be in luck. If programs on your system can do cursor positioning just by "printing" special character sequences, then these can be entered to Config just like those of a standard terminal. However, if programs have to directly address locations in the screen memory, then you have a problem and will have to do some programming. If you have an Amethyst, it is annoying but straightforward to write a terminal abstraction which handles your board (see the Terminal Abstraction Documentation). If you don't have an Amethyst, you'll have to add character-driven cursor positioning to your BIOS.

Fourth Step: A Question of Taste

There are several questions that the configuration program asks you for which there are no correct answers; rather, they are about your personal preferences. The first such question concerns the size of the swap file, as mentioned above.

The next question is the preferred cursor line. This line is the line of the screen that the cursor (Point) will be displayed on after a View Next Screen, Refresh Display, or similar command is executed. We like it to be just above the center of the screen (not counting the two reserved lines at the bottom).

The tab spacing, fill column and indent column values can be easily changed while you are running Mince. Thus, you are selecting the values that you would like them to have when you first enter Mince, and you need not worry about one or two special cases where you might want them to be different.

First you will be asked to select the tab spacing. Mince has semi-variable tabbing: tab stops are set automatically a fixed number of columns apart (for example, if the spacing is eight, tabs will be set at columns 8, 16, 24, ...). ANSI (American National Standards Institute) does not have a standard for this, but recommends an eight column spacing; this is also the CP/M standard. We prefer a five column spacing, however, especially for program source code.

Next is the fill column. Several commands deal with left and right margins for the text. Examples are Fill Paragraph and Center Line. The fill column is the right margin and is the first column in which characters cannot appear. We have found a value of 65 to be reasonable, although both 70 and 75 have been used in various documents. (This text uses 65.)

Related to the fill column is the indent column. It is effectively the left margin and is the first column in which text can appear. A value of zero is normal and causes the text to be against the left hand edge. Other parts of the documentation use indented paragraphs and an indentation of five.

The last "taste" question concerns the delay count. There are some things Mince does when it sees that the user has not typed a character for a little while, like echoing prefixes (e.g., "Meta:") or writing pages through to the swap file (that "Swapping..." message). The delay count tells Mince how long a "little while" is. We find that about four seconds' delay before auto-swapping is reasonable and a value of 350 results in that much delay on our system. Doubling or halving the value doubles or halves the delay.

Fifth Step: Testing

Now that you have told the configuration program about your terminal, you should select option 6: terminal testing. This should clear the screen, print columns of '*'s on the left and right edges, from the bottom up, then '*'s along the top and bottom from right to left. The text 'This should remain' and

'This should go away' will appear on the screen near the top, and the latter should be erased almost immediately.

If the '*'s do not properly ring the screen either the cursor positioning sequences are wrong or the size of the screen is improperly specified. If the text 'This should go away' is left on the screen the clear to end of line function is probably not working. In addition if some characters are missing or there are additional random characters, the padding might be wrong.

If the terminal test rolls the screen up one line, try reducing the number of columns by one (via option 5) and rerunning the test. If the test now works, you can restore the number of columns to what it was and Mince should work. The reason why the screen gets scrolled is that Config writes in the lower right character position but Mince does not.

Remember to install your terminal definitions (select option 7) on the swap file so that Mince will be able to find them later.

Sixth Step: The Finishing Touches

Now that you have told the configuration program about your system, you should go into Mince and try it out briefly. Does Mince seem to work properly? Are the delays about right?

After you are basically satisfied that everything went smoothly (don't worry about the fine points too much), you should make a backup copy (on a third disk -- not the original!) of all of the work that you just did. Do not forget to delete any stray files that you created when testing Mince before making the backup.

(Another fine point.)

If you have double density disks and would like to use them as double density, make the backup from the single density "master" to a double density "work" disk.

(End of fine point.)

You should now have three disks. The first one is the distribution disk. The other two are identical and contain the swap file as configured and Mince. Take the distribution disk and one of the copies and put them somewhere safe (and remove the "write enable" tabs).

You are left with one disk with a working Mince (of which you may want to make still more copies). We hope that you will enjoy using it.



This document is a series of lessons to teach the use of "Mince", a display-screen oriented text editor. The reader is assumed to have used a typewriter for the preparation of manuals, memos, or documents before, but never to have used a text editor. Additionally, the reader is assumed to already be familiar with logging on to the computer and typing at a computer terminal. Each lesson teaches a new set of commands or concepts about text editing in general and Mince in particular. There are eight lessons, on the following topics:

- Lesson 1: Getting Started
This lesson teaches about the cursor, typing in text, and the commands used to move around and delete and insert text.
- Lesson 2: Moving Around Faster
This lesson teaches commands for moving by words instead of characters, the universal repeat command, and beginning/end-of-line/sentence/buffer commands.
- Lesson 3: Reading and Writing Files
This lesson explains the computer file system and how Mince uses it to store text. Commands for reading and writing files are demonstrated. At this point, the reader will be able to use Mince effectively for simple document preparation, although document modification may be somewhat time-consuming still.
- Lesson 4: Searching
This lesson introduces the forward and reverse search commands.
- Lesson 5: Keyboard Culture
This lesson explains a few minor but useful Mince commands. Some common abbreviations used in the other Mince documentation are explained to allow use of the reference manual.
- Lesson 6: Killing and Moving Text
This lesson introduces the kill and yank commands. Extensive examples of text movement are given, since this is one of the more confusing aspects of Mince.
- Lesson 7: Text Processing Commands
This lesson explains the command used for "filling" paragraphs of text, and the commands used to set parameters which control this operation. Commands which change the case of words are explained.
- Lesson 8: Buffers
This lesson explains the effective use of multiple editing buffers.

Lesson 1: Getting Started

Mince is a text-preparation program for use with small computers. In many ways, it is similar to a very fancy office typewriter. However, it was designed to be used with a TV-screen computer terminal. During this first lesson, you should be getting used to typing on a computer terminal, if you haven't done some work on one already. The keys on each terminal or typewriter are placed a little differently; in particular, we will be making use of some of the out-of-the-way ones you don't usually hit.

So, let's get started. Sit down at the terminal attached to the computer, and type Mince's name to the computer. Type:

```
mince
```

and follow it with a carriage-return. (This is the key labelled "RETURN", "CR", or "ENTER" on your terminal.)

Mince will begin to run, and you will notice all sorts of things happening at once. There will be some buzzing over at the disc drives. This is where the computer stores Mince when it's not in use, and where you will store your text when you don't need it. (More about that in Lesson 3.) Your terminal screen will be blanked in preparation for your typing. A single line will appear at the bottom of your screen, something like:

```
Mince Version 2.5 (Normal) main: DELETE.ME -0%-
```

which tells you that Mince is ready and waiting to go to work.

Let's start by just typing something on the keyboard. Type the four words:

```
This is a test.
```

Notice that as you typed, what you were typing appeared at the top of the screen. At the end of the line you typed, right after the period, is a solid or blinking box or underline. This object, called the "cursor", is not something you typed; it is just an indicator of where you are in your text. It functions just like the carriage of an ordinary typewriter, showing you where the next thing you type will appear.

The next thing you type will be a carriage-return. As on an electric typewriter, hitting the carriage-return key puts you at the beginning of the next line. After the carriage-return, type another line:

I am testing this text editor.

and follow it with another carriage-return. Now you have a two-line document.

Notice that the line at the bottom of the screen has not changed position throughout this typing. The entire screen of a terminal does not scroll the same way that a paper would in a typewriter. Only the top portion of the screen (20 lines or so) will scroll upward, if you type enough text. This area is called the "window". Imagine it as a small viewing area onto a large document, and the name will make sense. Let's type a little more, just to get used to it. Type:

It's not much different from using a typewriter.

and type a carriage-return.

We could go on typing lines of text, but sooner or later we would make a mistake. Let's make one now, on purpose. Type just the two words:

But typewriterb

and stop there. So much for our "error". The "b" in "typewriterb" should have been an "s". What do we normally do on a typewriter if we type a wrong letter? We use the erase key if it's a fancy office typewriter, else (yecch!) white-out. Mince has an erase key, too. Type the key labelled "DEL" or "DELETE" or "RUBOUT". (On some computer terminals you may have to hold down the shift key to get a delete. Make certain that you really are going to hit DELETE and not some other character.) Observe that the "b" in "typewriterb" has disappeared from the screen, and the cursor has moved backward a space, to right after the "r". Type an "s" now, and the word "typewriters" will become correct. It truly IS very much like using a typewriter.

Suppose we realize that we had forgotten to type some word before the word "typewriters". Just hit the DELETE key enough times to erase the word "typewriters". (Go ahead and do that now.) Now type:

MOST typewriters

and stop. The line now reads "But MOST typewriters" and the cursor is right after the "s" is "typewriters".

We had seen that all text we type is entered at the cursor;

now we've seen that text we delete is deleted at the cursor as well. This rule always holds true in Mince, and there's another demonstration of it coming up.

What if, as before, we discovered a word missing, only on the FIRST line this time? For example, let's suppose we wanted to change the sentence reading "This is a test." to "This is only a test." by adding the word "only". You certainly could type the DELETE key enough times to erase all the way back, but it would be an awful waste of time if we had to do it that way.

The way we modify or add text in Mince is to "go" to the place in the text where we want text to be added or changed, and then type in whatever text we want there. In order to "go" somewhere, we position the cursor to that place on the screen. To get to the first line of text, we will move the cursor up three lines, one at a time.

Look on your keyboard for a key labelled "CTL", "CTRL", "CONTROL", "CNTRL", or the like. This is a very important key for Mince; it is called the control key. It is like a shift key: you hold it down, and while holding it down, you type one or more normal characters. Like a shift key, pressing it and releasing it has no effect. When you press the "p" key (DON'T do it now), you get a lower-case "p". If you press the "p" key while holding the "SHIFT" key, you get an upper-case "P". If you press the "p" key while holding the "CONTROL" key, you get something called a "Control P". It is a different variety of P, just like p and P, except instead of being an upper case letter or a lower case letter, it is called a "control character". All control characters are commands to Mince. (This makes the Control key very much like the "Code" key which is present on the fancy IBM typewriters.) You have seen what happens when you type normal upper and lower case characters; they go into the document. Control characters are used to control Mince, to manipulate the cursor and manipulate the text around it. (Note that although we will spell the control commands with a capital letter, it is not necessary to use the shift key AND the Control key to get the command. "Control P" is just the same as "Control p".)

To make Mince move the cursor to the previous line, we use the control character "Control P", the P being for "Previous". Do so: hold down the "CONTROL" key, like a shift key, and press the "P" key, while still holding the "CONTROL" key, then quickly release both. Now look at the screen: you will observe that the cursor has indeed moved up to the previous line, to a place right above where it had been.

Now let us go up two more lines, to the first line. Hold down the control key again, and type "P" twice while holding it. Again, the cursor will jump up a line each time.

Now that we are on the correct line for the change we wish to make, we can move around on that line to get to the correct

place. We can move backwards on that line by telling Mince to go backwards: this is done with a "Control B", B for "Backwards". Hold down the control key. Now watch the screen and press the "B" key several times, while still holding the control key. You will see the cursor move backwards, that is, to the left, one character position for each time you press the "B" key. Soon you will reach the beginning of the line, at which time you cannot go any further back, because that is the beginning of your text.

We now want to move forward to the place after the word "is", in order to insert the word "only". Hold down the control key, and type the letter "F" several times. "Control F" is the Mince command for "forward", i.e., move the cursor forward one character. While still holding down the control key, type F's until the cursor is under (or covering, on some terminals), the word "a". If you type the Control F too many times, simply type Control B's until you get to the right place. What you are doing now is the most important form of interaction with Mince - issuing commands until you are "at the right place", or "the right thing has happened", doing one command at a time and observing its effect, and repeating that process until you have achieved what you want.

Now the cursor is at the "a", and we wish to put the word "only" there. That is trivial: simply type the four letters o, n, l, and y. You will watch Mince move the rest of the line over to make room for the new text and you now have on that line:

```
This is onlya test.
```

with the cursor still under the "a". Immediately, we perceive a problem: there is no space between "only" and "a". This is simply because we did not enter one. Hit the space bar. Now we have:

```
This is only a test.
```

with the cursor still on the "a". We have now fixed the text we wished to. Note that in order to type in the new word, "only", we did not have to say anything special, we just moved the cursor to the right place, and started typing. Whenever we type a non-control character, it goes into the text at the cursor, and moves the cursor over one to the right. If you think for a little while, you will notice that that is just what it did when we were simply typing in the lines on the screen! It does look a little unusual (and not at all like a typewriter) to have an entire line moving to the right, but it's easy to get used to.

Now we must get back to the end of the document, where we had left off when we decided to add the word "only" to the first line. We can do this by going to the Next line, and the Next line, until we are where we want to be. We do this with the "Control N" command ("N" for "Next", as you might have guessed). Hold down the control key (As you have seen, if you do not the

characters you type will go into your text!), and press "N" three times. Each time, the cursor will move one line down. We are now on the right line. You will find that you are in the middle of the line, because Mince tries to keep you in the same "column" when going between lines. Type a few Control F's to get to the end of the line. Again, you will notice that at the edge of your text, the cursor stops, because you can go no further forward.

Now let's finish the sentence "But MOST typewriters are old-fashioned." Enter the words:

are old-fashioned.

and surely enough, your entire text appears correct before you:

This is only a test.
I am testing this new editor.
It's not much different from using a typewriter.
But MOST typewriters are old-fashioned.

If you haven't already done so, type a carriage-return after the last sentence. Now the text has five lines in it, and the last one is a blank line. Now type a "Control P" to move up to the previous line. The cursor is now at the "B" in "But".

Now type a "Control B". Notice that the cursor has moved to the end of the previous line, right after the period! This is not what we would expect from a typewriter, where the carriage would just have bounced off the left margin. In Mince, however, the carriage-return you type is turned into a "newline" character, which is inserted in the text. On the terminal's screen, this newline character looks like just that: a new line. But newline is a character, too, just like all the others you have been typing in the text. Since carriage-return or "newline" is a character just like any other, it can be moved across, just like any other character. Type a "Control F" now. As you probably expected, the cursor moves back to the beginning of the next line. Now type the DELETE key. We can delete the newline characters as well, just like any other characters. Now the text window should have:

This is a test.
I am testing this text editor.
It's not much different from using a typewriter. But MOST typewriter
old-fashioned.

Notice that the third line of text has gone past the edge of the terminal's screen, and the rest of the line appears on the next terminal line. (Probably the word "old-fashioned" is split across the two screen lines.) Just because there are two screen lines does not mean that there are two real lines of text there, separated by a real carriage-newline character, though. You can usually tell the difference by the fact that one line runs off to

the very last column on the screen. You will not usually type such long lines, just as you would not type off the end of the piece of paper in a typewriter carriage. It is possible, however, to get lines to join together and create a longer line, as we just did. Insert the newline back again: type carriage-return and it gets inserted in front of the cursor, just as we would expect. Insert another newline, so that there is a blank line inbetween the third and fourth (now fifth) lines of text.

What if we wanted to change "But MOST typewriters" to "Wood stoves"? We have to erase the first three words, then insert the new text. So far, the only way you know to delete text is to use the DELETE key. But if you were to hit it now, it would delete the character you just entered, that is, the newline. We could, of course, type some Control F's to get the cursor past the words, and then type some DELETES. (Or we could remove the words one character at a time, by alternately typing Control F and DELETE.) But this would be cumbersome. Instead, Mince has a command to delete the next character in the text rather than the previous one: "Control D". Hold down the control key and type "D" enough to get rid of the words "But MOST typewriters". Notice that, as usual, the entire line is "eaten up" and moves to the left as the characters disappear.

Now that the words are gone, type the DELETE key, and the blank line we inserted earlier will also disappear. We see that the DELETE key erases characters to the left of the cursor and the Control D command erases characters in the other direction, that is, it erases the character that the cursor is on. Now type "Wood stoves" to insert those words at the beginning of the line. Type a space after "stoves" if you happened to delete the one to the right of "typewriters" earlier.

Take a quick look at the "mode line" at the bottom of the screen again. The percentage mark is no longer zero, as it was when we started. This number just says roughly how far through the file the cursor is. It's not too useful when you can see your entire document on the screen as we can now, but for large gobs of text, it's handy to know where you are. Also, there is a star at the right edge of the mode line which appeared there when you started typing. This asterisk means that the text which is on the screen is different from what you started with (in this case, nothing at all). It turns on in order to tell you that your text has not been saved anywhere, and that if you leave Mince without saving it, your work will be lost. We'll learn about saving the text we type in Lesson 3.

You may want to continue experimenting with the things you have just learned: when you are done, you will have to leave the editor: this is done as follows: type a "Control X" followed by a "Control C" (C for "Command level"). This is not an easy command to type or remember. That is reasonable, though, because you may be editing for hours, and you will only type it once. Hold down the control key, and type an "X" and a "C" while holding it.

Mince will respond:

Abandon modified buffer(s)?

because you have done work that you have not saved. That's O.K. for now. Type "Y". Mince will leave you at command level, at the bottom of the screen.

We just learned the six most important Mince commands:

DELETE	Erase the last character.
Control D	Erase the next character.
Control F	Go Forward one character.
Control B	Go Backward one character.
Control N	Go to the Next line.
Control P	Go to the Previous line.
Control X Control C	Quit the editor.

We have also learned some basic Mince vocabulary:

cursor: The solid or blinking box or underline which shows where you are about to insert or delete text on the screen.

window: The area which covers most of the screen, and which displays twenty to twenty-five lines of the text which you are currently editing. This window moves around so that the cursor is always somewhere in the window, regardless of how you move it while inserting or deleting text.

mode line: The line at the bottom of the screen which tells you that (1) you are talking to Mince, (2) how far through the text you are, and (3) whether the text on the screen has been changed since it was last saved. There are quite a few other pieces of information which the mode line gives; more about those in Lessons 3 and 8.

Practice using the commands in this lesson until you are thoroughly familiar with them. They are the most important and useful commands. With these commands, and one or two others, you can do just about anything you will ever be called upon to do. All the other commands just make it easier to do more complex things, but you can always move around and type in text with these.

Lesson 2: Moving Around Faster

To start this lesson, enter Mince as you learned in Lesson 1. Type the following text in to Mince. Since it is poetry of a sort, be sure to break the lines (type carriage-returns) at the proper places. Use the DELETE key to correct some of the mistakes as you type it in, but don't be too careful about the words' spelling -- the poem is intentionally nonsense. The text:

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe.
All mimsy were the borogoves,
And the mome raths outgrabe.

"Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird and shun
The frumious Bandersnatch."

by Lewis Carroll

Now let's go back and make certain that the text IS letter-perfect. Do you have blank lines between the verses and before and after the author's name? This should be a good chance to review using Control P (Previous line), Control N (Next line), Control B (Backward character), Control F (Forward character), and Control D (Delete character). It won't take long to make it perfect -- it's easy to turn a draft into a final if you don't have to retype the whole thing!

As an aside... How are you typing the control characters? Using the index finger of one hand to hold down the control key and using the index finger of the other hand to hit the letters? That's the way most new users (and programmers!) do it. But remember that you can go much faster if you learn to touch-type the control keys, too.

Well, as usual, we are going to change the text you just typed in and perfected. (With any luck, after we're done, the first verse will make a little more sense!) Before we start, make certain that you typed a newline as the last character, and move to the end of the text. The cursor should be directly under the "b" in "by Lewis Carroll". Changing this verse could be a

real chore if all we had were the commands we learned in Lesson 1. Moving one character at a time is slow and fairly annoying, especially if we can see the place we want to get to to make a change.

For example, how would you change the last word on the previous line? Typing Control P and lots of Control F's could take a while. (If you're a smarty, you have realized that it would be easier to just type a Control B and pass over the newline character! But then what about the last word on the second line above that?) To illustrate a new command, let's change the word "Bandersnatch" to "Sandpiper". Type some Control P's to position yourself on the proper line. Now type a Control E. Control E stands for "go to the End of the line", which you should have seen the cursor do! Now type in enough DELETES to erase the word "Bandersnatch" (Perhaps you will want to type some Control B's to pass over the punctuation first.) and then type in "Sandpiper".

Now, suppose we wanted to change the word "The" at the beginning of this line to the word "A". We could type some Control B's, but this would take a few too many keystrokes. Instead, there is a Mince command which moves to the beginning of the line: Control A. It doesn't stand for anything in particular, but an easy way to remember that a Control A moves to the beginning of the line is that "A" is at the beginning of the alphabet, or that it is on the far left of the keyboard and the cursor moves to the far left of the line. So, go ahead now and type a Control A, then type three Control D's to delete the word "The" and then insert the word "A". Now type enough Control N's to get back to the bottom of the text. Notice that if you keep typing control N's, you still stop at the last line you entered. Similarly, if you type Control P's when you are at the beginning of your text, you will just stay in the same place. (Similarly for Control B's at the beginning and Control F's at the end.)

Now that we're at the end of the text, what if we noticed that we wanted to change the line that says "All mimsy were the borogoves"? We could type a lot of Control P's, one at a time. This is easy enough but may be more time-consuming for longer "distances". The command Control U (for "Universal repeat count") will allow us to make the jump with less typing. Type the Control U. You will notice a message at the bottom of the screen saying "Arg:" appear, along with the number "4". Type a "9", (The number in the message will change to "9" also.) since the line we want is nine lines above where the cursor is now. Then type a Control P. Nine Control P commands in a row will automatically get executed. If we decided that this wasn't where we wanted to be, we could have typed "Control U 9 Control N", and gone down nine lines just as easily. Control U repeats anything. Anything? Yes. Type "Control U 10 *" (that is, type the Control U, then type the numeral one, then the numeral zero, then an asterisk). Ten stars will appear in your text. Now type a Control U, then type the number ten, and then type the DELETE

key. They will all go away again. Control U really does repeat anything.

Why did the number four appear as the "Universal" repeat count each time before we typed in our number? If we hadn't typed a number, whatever character (control or otherwise) typed subsequently would have been repeated four times. This is just for convenience: for example, it is easier for most people to type "Control U, Control P, Control P, Control P" to move back six lines in the text than it is to reach out to type the number six after the first Control U. This isn't quite so true for large numbers, so Control U has another unusual property: it multiplies any other previous universal repeat given! This means that if we type Control U in front of another Control U, the number of times whatever command we type next will be repeated is 16. In the same manner, "Control U 7 Control U" will repeat the next command 28 times. This can be quite handy, since "Control U Control U Control N" moves the cursor about two thirds of a screenful on most terminals. (That sequence is typed about as often as "Control U Control N Control U Control N", which moves down eight lines. You really can almost always avoid typing a number for the Universal repeat!) Try that now, on the line we are on: type "Control U Control U Control F", to move the cursor forward sixteen characters.

We can jump across characters faster than before just by using the universal repeat command. But it would be much more convenient to be able to move across words without counting how many characters there are in each of them. For that, we will have to learn about a new key on the keyboard and a way to give Mince commands other than the control key.

Search around on your keyboard for a key labelled "ESC", "ESCAPE", "ALT", or "ALTMODE", or something like that. This is called the "ESCAPE" key, which generates a character called the "ESCAPE character". We will refer to it by the letters ESC; this means strike and release the ESCAPE key, do not type the letters E, S, and C! This key does not work like the Control key; you do not hold it down while typing characters. Just type it and then type another character after it. For example, ESC F would be typed by first hitting the Escape key, then hitting the letter "f".

Try that now. Type an "ESC F" and see what happens on the screen. Try it again a few times. The ESC F command stands for "move Forward a word". Let's change the word "brillig" to "overcast". Move to the top line of the screen (in whatever method is most convenient for you now), to the first line of text, on the left-hand edge (use Control A to do this). Use the ESC F command to skip past the word "'Twas", and then use it again to skip past the word "brillig". Two interesting things happened when you typed those ESC F's. On the first one, we had punctuation in front of the word; the ESC F command doesn't care, and just skips to the end of the word anyway. On the second one,

the ESC F stopped before going past the comma; this is because the punctuation is not part of the word. What the ESC F command really does is to move to the end of whatever word is in front of it. If we were in the middle of a word and typed ESC F, it would go to the end of that word. If we were five lines above a word, but all five lines were blank, ESC F would still put us at the end of that word.

To delete the word, we could type many DELETE's. But as you might have guessed, there is an ESC command to do that, too. To delete words, use the ESC DELETE command. Type the Escape key, then type the DELETE key, and the word "brillig" will disappear. Now type in "overcast".

Let's also change the word "toves" to "foxes". To do that, move past the next three words (only THREE, the words "and", "the", and "slithy") with the ESC F command. Now let's delete the word "toves", by using the ESC D command. As you can now see, there are parallel commands for Control F (ESC F), DELETE (ESC DELETE), and Control D (ESC D), all of which work on words rather than characters. Now that we've deleted the word with ESC D, let's type in the word "foxes".

But look at the screen! Since we didn't type a space before the next word (or maybe you noticed that there wasn't one, and did!), the last two are run together. Why did the space between the words get deleted with ESC D, but not with ESC DELETE before? Just like the ESC F command, the ESC D command only knows about deleting to the end of the next word. So, if you are in front of a space which itself is in front of the next word, ESC D will delete the space, too, since all Mince does is to delete until it reaches the end of the next word. So, in the example mentioned above with five blank lines and then a word, if we had typed ESC D rather than ESC F, we would have deleted the five blank lines, and the word following them.

Let's fix up the missing space by typing Control B's until the cursor is on top of the "f" in "foxes", and then typing a space. You may have noticed that we didn't mention an ESC B (to go with Control B) above. Well, surely enough there is one. Type that now, and notice that the cursor jumps backward across the word "slithy". Let's delete that word (either with Control D's or ESC D) and type in the words "quick brown". Just so you know that you've gotten everything right this lesson, the first line should now read "'Twas overcast, and the quick brown foxes". If it's not correct, take a minute to fix it up now using some of the commands you've learned in this lesson.

Now let's change the word "wabe" to "swamp". One easy way to do this would be to notice that the word "wabe" is at the end of the next line, almost below where we are now. It could be reached by a single Control N. Another way would be to see that it is the eighth word after the ones we just typed in, and that it can be reached by typing Control U 7 ESC F. However you get

there, delete the word using either ESC D or ESC DELETE, and insert the word "swamp". If you used the ESC DELETE command, you might have to type the period at the end of the line in again, since it may have been eaten up in the backward deletion of the word.

Suppose we wanted to insert the name of the poem, "Jabberwocky", in the line with the author's name? We could type some Control N's to get to the last line, but there is a quicker way. Type "ESC >" (the ESCAPE key, then a greater-than symbol). This, as you see, puts the cursor at the end of all the text. Now type a Control P, to position the cursor at the beginning of the proper line, and insert the title and a comma. There is a command to move as far as possible in the other direction, too. Type "ESC <", and you will move the cursor to the beginning of the text.

There are two more quick-motion commands which you may or may not want to practice using. These are ESC A and ESC E. They are similar to Control A and Control E, except that they move to the beginning and end of sentences rather than lines. Now we have a reasonably complete set of parallels for the control commands we learned in Lesson 1:

Control F	Move forward a character
ESC F	Move forward a word
Control B	Move backward a character
ESC B	Move backward a word
Control D	Delete forward a character
ESC D	Delete forward a word
DELETE	Delete backward a character
ESC DELETE	Delete backward a word
Control A	Move to beginning of line
ESC A	Move back to beginning of sentence
Control E	Move to end of line
ESC E	Move forward to end of sentence
ESC <	Go to beginning of text
ESC >	Go to end of text

Play around with these commands a little while, until you are used to what they do and are able to remember most of them off the top of your head, without looking at this page. We haven't yet succeeded in making sense out of the first verse; now's your opportunity!

When you are done and want to leave the editor, type

"Control X Control C", as we did in Lesson 1. We'll learn more about what this and some of the other Control X commands are used for in the next lesson.

Lesson 3: Reading and Writing Files

In this lesson, we won't be using Mince at all. (How refreshing, eh?) We'll just be chatting about computer "files". At the end of this lesson, you can try out the commands you learn, and you will (finally) know all the commands you need to do some "real work" using the editor.

On to files... A computer's filing system is pretty much the same as the one you or I use. If you want to look at an old report, what do you do? You pull out the file from the filing drawer and put it on your desk. To find a particular spot in the report, you page through it on your desk. You might change it or add something to it, then put it back into the file drawer.

The operations are similar on the computer system. The only differences are the ways in which the "file drawers", "files", and "pages" work, and in the names computer scientists decided to give them.

Let's think of the disc drive as the filing cabinet and each of those flexible "floppy discs" as a sort of removable filing drawer. The unfortunate property of these discs is that they are only useful when put into the disc drive, which is not true of filing drawers and cabinets! On each of the discs, as in each of the drawers, are many files. You can see what files you have in a drawer either by leafing through them, or by looking at the short label on the front of the file drawer. There is a label space on each flexible disc which is similar to the one on the whole drawer, but how can you find out the names on each of the files? There is a computer program to do this for you. Just try typing "DIR" to the computer system when you aren't using Mince, and it will list the names of the files. Most files have two names, like "PRGINTRO.DOC", that is, names with some characters, then a dot, and then some more characters. The first bunch of characters can't be longer than eight, and the second bunch can be up to three characters long. This is a lot more limited than what you can write on a file folder, so you can now understand why some computer files seem to have such cryptic names.

What is the analogy of getting the file out and putting it on your desk? To Mince, this is called "reading" a file. Mince has an unusual way of manipulating it, though. Just as you would probably not mark up the originals of a report in your file

folder, Mince does not change the contents of the file on the floppy disc. You would make a photocopy of your work, mark it up or change it, and then perhaps retype it and replace what was originally in the file folder. Mince does something similar: it "reads in" a copy of that file, and then lets you modify that copy. This has one unique drawback and one unique benefit: if you make a mistake, you can just quit, and you haven't actually wreaked havoc on the original file; on the other hand, if you don't remember that you must save what you've done back in the file, you can lose all your hard work.

Therefore, there is a command in Mince to save the work that you've been doing as the original again. This is analogous to putting your new work back into the file, and back into the file drawer. To Mince (and computer scientists), this is known as "writing" the file. (It surely isn't what you or I think of when someone says "write the file out". "Really," you would say. "Longhand?")

In Mince, the part which performs the function of the working desk while we manipulate the file is the "buffer". Files are "read in" to the buffer, and they are "written out" back to the computer filing system. The computer terminal screen is a sort of window into that buffer, and we can move the window around with the control commands you have been learning recently. Just as paper files have pages, our computer file has pages, which we can see by issuing "page" commands to Mince.

Enough generalizing. Here are the commands:

Control X	Control R:	Read the file from the computer filing system into the Mince buffer.
Control X	Control W:	Write the file out from the Mince buffer back into the computer filing system.
Control V:		View next page of the file in the buffer.
ESC V:		View previous page of the file.

Actually, the Control V and ESC V commands are misnomers. They really have nothing at all to do with pages of text as we would ordinarily see them on a desk. They just serve to move the window so that most of it rests on what was unseen text, either before or after the windowful that is currently in the window. After all, most paper pages have 55 to 60 lines of text, and our terminal screen window has only 20 or so lines. In order to provide some continuity between screenfuls, these commands will overlap pages: one or two lines at either end of the screen will be from the next screenful of text in that direction. The View Screen commands take Universal repeats, just like other commands, so that "Control U 5 Control V" will move the window "down" in

the text by five screenfuls, or about 100 lines. These commands do not do anything that can't be done with Control P and Control N; they just do it a little faster.

The Control X commands are a bunch of commands for which there was no room in control keys for, or commands which we want to be absolutely certain we want to type, so that there can be no mistake about what we are doing. Remember the "Control X Control C" command? Now that you know it has the possibility of throwing you out of Mince without your having saved the work you've been doing, aren't you glad it's just a little harder to type? (Remember back in Lesson 1 when we typed Control X Control C? Mince asked us if we wanted to "abandon" a "modified buffer". Now you know what that's all about!) The easiest way to remember what the Control X means is to imagine that these specialized commands are part of the "extended" command set.

The "Control X Control R" command, in order to read a file, must know which file you want to read. When you use it, it will ask you, in the echo line (the last line on the screen), for the name of the file you wish to read into Mince's buffer. Just type in the file name, followed by the carriage-return key. Don't worry about remembering all this -- when you type "Control X Control R", Mince will print out a message asking you to type in the file name and reminding you that you have to enter it with a carriage-return. It looks something like:

File to Read <CR>:

Similar things happen with the "Control X Control W" command. In general, we will want to write out the buffer back to the same file we read it in from, but occasionally, we will want to save it in a new file. (This is particularly true of a form letter that we modify slightly for a particular recipient, and do not want to change the original for.) After you type "Control X Control W", Mince will say:

File to Write <CR>:

You can again type in the name of the file into which the text is to be written, followed by a carriage-return. If you make mistakes in the name of the file, you can use the DELETE key, just as you would in the editor.

Since writing files back out to the same place they were read from is such a common operation, Mince allows you to avoid typing the entire file name each time you give the Control X Control W command. Rather than typing out the file name, just type a carriage-return. Mince will write the buffer out to the computer filing system under whatever name was last used in either a Control X Control R or Control X Control W command. How do you know where it will go without always remembering the file name you last used? Mince displays that name in the "mode line", which we learned about in Lesson 1. It looks something like:

Mince Version 2.5 (Normal) main B:PRGINTRO.DOC -0%-

and the phrase "B:PRGINTRO.DOC" says what the default file read/write name will be if we do not specify it; that is, if we just type a carriage-return in response to any of the file name questions. (You may have noticed the "B:" alongside the file name. This is due to the fact that computers, like people, have many filing cabinets. Just as you would tell someone that a file is in the "right-hand" filing cabinet, you must tell the computer which disc drive to look for the file in or to store it in. The computer disc drives you will be using are named "A", "B", etc.)

Some kinds of files get pretty large. Computer files are no exception. Computer scientists just measure them in thousands of characters rather than in inches of paper. If you had a very large file to look at, you would never fit each page of paper side-by-side on your desk so that you could see all of them at any instant. Instead, you would probably keep 5 or so pages spread out on your desk, and keep the rest in stacks, occasionally removing or adding a page on a stack.

Similarly, many many thousands of characters cannot all fit into your computer at once. (Usually, Mince has room for ten to twenty thousand.) Therefore, your buffers of text may not be completely residing in memory at once. Mince has a special file on disc (not related to the other files we've been talking about) which is similar to the stacks of paper you would keep on your desk. As you use each new page (thousand characters) of the text buffer you are editing, Mince will pull it out of the "stack of pages" by reading it from its special "page file".

If you go back to old pages or add new text to pages, they will generally be in memory rather than on the stack in the page file. But, occasionally, the computer's memory will fill up, as would your desk if you kept looking at more and more pages. So, occasionally, Mince must write some of its old pages back to disc in order to make room for new ones.

You will know what is happening by the click or buzz at the disc drives. There will also be a message printed at the lower right of the screen "Swapping..." which means that Mince is exchanging a page on disc for a page in memory. This message will go away when the swap is complete.

Why do you need to know all this? Well, occasionally you may have to wait for this page swapping to happen. Why? The computer stops listening to the terminal keyboard for a short time when it swaps pages. In particular, when you are just beginning to work on a buffer of text which you read in via a Control X Control R command, pieces of the buffer will be paged in as you need them. This means that Control V's will occasionally take a little longer than usual and produce the "Swapping..." message. Once read in, though, the pages you use

most (e.g. modify or go back to) will remain, and "page waits" will be less frequent. (Amazingly enough, computer scientists call this concept "paging" too! But a page of 1000 characters has little to do with a 55-to-60 line page of paper or a 20-line text window on a buffer!)

If you stop using the keyboard for a little while, Mince may spend a little time "cleaning up" while you are idle. It will swap out pages you have modified, so that any page swapping which is done while you are actually doing work later will occur a little faster. Don't worry about this too much; when you start typing on the keyboard again, the intermittent housecleaning noises will go away.

Don't worry about trying out the commands you have learned in this lesson, unless you have to start to work using the text-processing system right away. In the next lesson, you will get a chance to try out "Control X Control R" and "Control V" and "ESC V".

Lesson 4: Searching

For this lesson, you will be reading from your terminal rather than from this paper. Get into Mince as usual, and type:

```
Control X Control R lesson4.doc
```

followed by a carriage-return. You should see this text appear on your screen. If it does not, and Mince says "New File" in the echo line at the bottom of the screen, first make sure that you had typed the name correctly. If not, just try again; if so, stop and get help finding the file.

We will be playing around with some new commands on this text. In order to keep reading from your terminal, you will have to type Control V's (View next page) every time you reach the bottom of the screen. Do that now. Type a "Control V" and forget about the paper copy of this lesson.

The first command we will learn this time is the "search" command, "Control S". Suppose someone has given you a draft for corrections, and somewhere in the middle has marked a phrase to be changed. It's easy enough to see it on paper, because the place is marked in red ink or the like. But it requires extra work to find it on the computer. It's almost as though you had been handed the same report, unmarked, and been told that some phrase should be deleted!

Certainly we could scan through the file looking for the marked place using Control V's, but it might take a while. Instead, we can use the computer to search through the file, looking for the proper place.

Try that now. Type:

```
Control S
```

(You might want to look back at the paper copy of this lesson for just a little while now, since Control V's won't work while we're in the middle of a search.) Mince will respond to the Control S by printing "Forward Search <ESC>:" in the echo line at the bottom of the screen. Now type in the word:

```
search
```

You will see it echoed at the bottom of the screen. If you make a mistake while typing this, you can use the DELETE key, just as you would normally. After you've entered the string to search for, type the ESCAPE key. This tells Mince to begin searching.

A forward search begins at the character the cursor is on, continues until the searched-for item is found, and when it is, positions the cursor just after it. So, when you hit the ESCAPE key, you should have seen the cursor jump to one of the occurrences of the word "search" in this lesson. It should still be there now, unless you've had to type Control V's to read more. If so, try it again. Type:

```
Control S  search  ESCAPE
```

Now just try typing:

```
Control S  ESCAPE
```

That is, do not type in any character string to be search for; just hit the ESCAPE key right after the Control S. You should notice the cursor jump to the next occurrence of the word "search". When you do not give the Control S command a new character string to look for, it searches for whatever you told it last time. This can be quite a timesaver, especially when working with technical papers, where long, hard-to-type phrases are commonly used several times, and you want to find, say, the sixth occurrence.

What happens if there is no character string exactly like the one you type for Control S? Try it and see. Do a search for a nonsense word. If there is no such word between the cursor and the end of the text, as you see, Mince will cause the terminal to beep and will print the phrase "Not Found" at the right of the echo line.

As you may have guessed by the message that Mince printed in the echo line earlier, "Forward Search", there is a "backward" search as well. It is used to search backwards in the text from the current cursor position. The command is called "Reverse search", and is performed by Control R. All of the features of Control R are identical to Control S, except that, since it searches through the text in reverse, it leaves the cursor before the closest item that matches the string, rather than after it. Notice that the character string to be searched for is saved and is the same one for both Control R and Control S. So, you may find yourself typing "Control S", a string, and the ESCAPE, and then, if you hear a beep, typing "Control R ESCAPE" because you know the string is in the text somewhere.

Play around with these a little bit, then type Control X Control C to leave the editor. The next lesson, Lesson 5, should be read off the printed page, but for Lesson 6 we'll be working exclusively on the terminal again.

Lesson 5: Keyboard Culture

This lesson is a short question-and-answer session. It explains some useful details about the other Mince documentation, as well as some typing conventions used through the rest of this document, the Mince User's Manual, and in Mince's screen display.

(Q) I looked at the user's manual and noticed that Control commands are spelled "C-". Why?

(A) There are three common ways of representing control characters:

- (1) Control P
- (2) C-P
- (3) ^P

The reason for using #2 is that it is shorter to type and easier to discern from surrounding text than #1. That is the format which will be used throughout the rest of these lessons. It is also the format which is used in the Mince User's Manual. Format #3 is used in Mince screen displays. If you had a C-P (Control P, remember) in your text, and representation #2 were used, there would be no way to tell it from the three characters C, -, and P all in a row. The "^" (caret or uparrow) character is much less used, so it is a better representation for use with general text; it stands out almost as much as "C-" and is not quite so likely to be in any text you encounter.

(Q) I thought all control characters I typed were interpreted as commands to control Mince. How on earth could I get one into my text?

(A) With the C-Q command. C-Q stands for "Quote next character". So, typing C-Q C-P would put a C-P into your text buffer. (It would display on the screen window as "^P", remember.) Exactly why you would want to insert control characters is another good question; we don't have any answer for that!

(Q) In the User's Manual, the Escape commands are abbreviated "M-". Is that for the same reason as "C-"?

- (A) Basically, yes. The reason for "M-B" being the same as "ESC B" is just a bit more obscure. Once upon a time, someone thought that it would be neat to have yet another Control key, which also worked just like the shift key. They went ahead and made a keyboard which had one; it was called the "Meta" key. (One major terminal manufacturer even offers such a key today.) Programmers who see it think it's "neat"; it's VERY untypewriterlike, though. So, those of us without a Meta key for Meta-commands simulate it by typing an Escape in front of the command character. From now on in these lessons (and in the other manuals) we will use "M-" to be the abbreviation for prefixing a command with the ESC character. So:

```

C-K    is Control K
M-K    is Meta-K, typed ESC K
M-C-K  is Meta-Control-K (!) and
        is typed ESC Control K.

```

These conventions will make it easier for you to read through the text of lessons and manuals more quickly. You will also notice that if you type the Escape key and wait a second, you will see "Meta:" appear in the echo line. This occurs so that you don't forget that you have typed that prefix character.

- (Q) Suppose I type C-U and then decide that I really didn't want to repeat any command. What should I do?
- (A) Type a C-G. The C-G command flushes all prefix characters. It works just as well with Escape (when you decide you don't want to do a Meta command), C-X, and C-S/C-R (where it cancels the type-in of a search string). C-G makes your terminal bell beep; this is your assurance that the command has worked. Typing C-G just makes the terminal beep otherwise. This could be useful if you give Mince a series of advanced commands that will take a while. You can just type a C-G after all of them, and Mince will beep when it's all done.
- (Q) That's interesting. You mean I can type things without waiting for the others to happen? I hate waiting for the screen to redisplay the window!
- (A) Absolutely. This speeds typing up for you; it's called "type-ahead". For example, if you know that after the next occurrence of a particular word you want to enter a comma, you can type C-S, the word, an ESC, and a comma immediately thereafter. You don't need to wait for the cursor to jump or the screen to redisplay if you don't want to. Just make sure that what you've done is correct, though. If you type a

C-V immediately after that, you might never get to see the screenful of text after it had been modified. Mince assumes that if you are typing very quickly, you know what you are doing, and would only be annoyed by the flicker and flash of a constantly changing screen. If you've already gone somewhere else by the time the current screenful of text is changed, Mince won't bother with all the flashing. It just always tries to show the screenful of text where the cursor is positioned.

- (Q) Great. Then why was there that big explanation of the "Swapping..." message in Lesson 4?
- (A) *sigh* That's the one exception to the rule. When Mince uses the disc drives, both to read or write files and to read or write pages of a file, it ignores the keyboard. That is an unfortunate result of using this particular sort of small computer.
- (Q) Whenever I want to insert text, I move to the right place in the text and start typing, right? But it's really annoying, when I want to insert whole sentences or paragraphs, to see the line where I'm entering keep moving text over to make room. What do you do about that?
- (A) One possible method would be to insert a carriage-return before you start entering text. That way, you will always be typing text on a blank line, and the redisplay won't occur. Unfortunately, if you insert a carriage-return, then start typing, you will still be on the same line you wanted to avoid in the first place! You must insert the carriage-return, then type a C-B to get back to where the blank line is. There is a command to do this automatically; C-O (for "Open lines") will insert the newline and place you on it to give you a clear line for text entry. From then on, you can just use carriage-return, as you normally would.
- (Q) Why do C-S and C-R and the C-X commands put the termination character abbreviations in <...>'s?
- (A) Most things Mince displays in angle brackets are specially labelled keys on your keyboard.

<CR>	the Carriage-return key
<ESC>	the Escape key
	the Delete key
<LF>	the "Linefeed" key

We also have one called <NL> which you may run across. It stands for "New Line". There is no such key on the keyboard. But if you want to search for one word at the end of a line

followed by another word at the beginning of the next line, you would type C-S, the first word, a carriage-return, the second word, and then the Escape key. In the echo line, you will see "<NL>" where you had typed the carriage-return, to tell you that the search will be looking for a new line. inbetween the two words. Just as with typing regular text in the window, the Delete key will erase newlines, just like any other character.

(Q) Thank you for answering EVERY question I could EVER CONCEIVABLY ask about Mince.

(A) Obviously, we haven't even scratched the surface here. If you have questions or forget what certain commands do, try looking in the Mince User's Manual first. If there are no good answers there, give us a call or drop us a note in the mail.

Lesson 6: Killing and Moving Text

During Lesson 6, you should be reading from your terminal again, rather than from this manual. Type to the operating system:

```
mince lesson6.doc
```

and wait for Mince to get started. By putting a file name on the command line, you cause the file to be automatically read in when Mince starts up. As with Lesson 4, if Mince says "New File", and this text does not appear, something is wrong. Determine the cause and use the C-X C-R command to read in the file, if you need to.

In this lesson, we are going to learn how to delete whole bunches of text, rather than characters or words at a time. The command to delete lines is C-K. (Control K stands for "Kill line".) Move the cursor to the first line of this paragraph with C-P or C-N. Now type a C-K. Whatever was on that line has disappeared. Now type another C-K. The line itself has disappeared and all the other lines of text have been moved up.

Now move the cursor to the first line of this paragraph, and move halfway into the line with C-F's or M-F's. Again, type a C-K. You will notice that only the part of the line to the right of the cursor was killed. C-K kills text starting with the character which the cursor is resting on until it reaches a newline, or the newline itself, if the line is blank.

You see that in order to physically remove an entire line AND its contents, you must type two C-K's. Similarly, C-U C-K removes two entire lines and their contents, rather than four. You will find this command useful for retyping previously existing lines of text, where you want to remove what the line (or the "rest" of the line, starting at the cursor) says, but want to fill it with something else.

Typing many C-K's in a row could get to be a drag, especially if you intend to delete an entire paragraph or chapter. You could count the lines to be deleted and type C-U, twice the number, C-K; this would still be pretty time-consuming. (Counting lines is even more annoying than typing C-K many times!) Instead, Mince has a command to kill an entire region of text. It is called C-W (for "Wipe region").

You must define the region of text to be killed first. One end of the region which C-W will wipe out is shown by where the cursor is in the text. The other end of the region is given by a "mark", which we will now learn how to set. The command which sets the invisible mark is C-@ (for "set the mark AT this position"). Move the cursor to the beginning of some line in the middle of this paragraph. Type C-@. (Note that on some terminals you may have to hold down the Shift key and the Control key to get a C-@.) Down in the echo line you should see the message "Mark Set." If it does not appear, you may have a poor terminal whose C-@ doesn't send the command. Try typing C-@ again, and if it doesn't work this time, type a Control SPACE now, and use it from now on if it works, rather than the Control at-sign. If the "Mark Set" message still hasn't appeared down in the echo line, you will have to type ESC SPACE instead, and use the M-<SPACE> command from now on instead of either C-@ or C-<SPACE>.

Having figured that out, move to the end of the same line and type a C-W. Look at the text; the portion of it between the cursor (at the end of the line) and the invisible mark (at the beginning of the line) should have disappeared.

Let's wipe something slightly bigger than a line. Move the cursor to the beginning of this paragraph, and set the mark there by typing C-@. Now move the cursor to the end of this paragraph and type a C-W. You shouldn't be able to read this sentence any more!

What if we make a mistake and wipe out a huge block of text unintentionally? There is a command to retrieve the text which was just killed, C-Y (for "Yank back killed text"). Do that now: type a C-Y and the paragraph we just deleted should appear back where the cursor is now.

Now move the cursor down a few lines and type C-Y again. Another copy of that text appears at the cursor. Type the C-Y again; the text should be replicated one more time. You've just learned how to copy or move text, all with one command, C-Y! To make a copy of some text, wipe it out (using C-@ and C-W) and immediately yank it back with a C-Y in that spot (i.e., without moving the cursor). Then, move the cursor to the spot where you would like the copy to appear, and type another C-Y. If you want to move a block of text rather than copy it, just don't type the C-Y at the original position. (If deleting text and then yanking it back in the same place in order to do a copy of some text makes you nervous, you might want to look at the description of the M-W command in the Mince User's Manual. It works similarly to C-W, but it does the first C-Y automatically for you.)

C-W is not the only command which saves text in case you want to yank it back. C-K, M-D, and M- all save text as well. In general, if you delete anything larger than a

character, Mince will save it for you in case you want to move or copy it (or undo a mistake!).

As an example, move the cursor to the beginning of this paragraph, and type a couple of C-K's. Notice that a "plus" sign has come on at the right edge of the mode line. This means that if you continue to type C-K's, the text killed will be added on to whatever killed text is already being stored. So, you can kill a region of text, with either a C-@/C-W or with a bunch of C-K's (or even M-D's or some combination of all of these), and a C-Y will still yank the entire region back.

Mince will only store your "most recent" bunch of text kills, however. What determines what is "recent" and what is not? The plus-sign at the right of the mode line. If you are about to delete something larger than a character and the plus is not there, you will be throwing away whatever previous bunch of kills you did. In general, all this amounts to is that, if you give any movement commands or insert anything after killing some text, you will "close off" the current group of kills. (You will also see the plus-sign go away.) Any C-Y's you type after this will retrieve that group. Any C-K's, M-D's, M-'s, or C-W's you do after this will throw away that group of kills and start a brand new one.

There is a command to "turn on the plus sign". This is used if you want to move groups of lines from several different places all to one place. Certainly you could do this manually, doing a few C-K's, moving to the right place, doing a C-Y, going somewhere else and doing some C-K's or a C-W, moving back to the right place, doing another C-Y, etc. It would be much easier to do C-K's or C-W's in all the various places and then yank the whole thing back with a single C-Y. But we said earlier that movement away from the place of the text killing or wiping causes the plus-sign to go off and the current bunch of kills to be "closed off". The command to turn the plus sign back on is M-C-W. (This is your first "M-C-" command. Remember that you just type ESC Control W.) You can remember this command by its intimate relation with C-W (and M-W, if you use it).

As M-C-W is sort of hard to think about, there is no substitute for practice. Move to the beginning of this paragraph and kill the text on the first line, in whatever manner is convenient. Then move to the last line. Notice that the plus in the mode line has gone off. Type a M-C-W to turn it on, then kill off the text on the last line. Now move into the middle of what's left and type a C-Y. You will see both the first and last lines appear there at once.

Play around with all these commands a little bit. Try deleting the last five words of this sentence and yanking them back at the beginning of the sentence. Then, move this paragraph so that it is the first paragraph in the lesson. Try doing C-W's with the cursor both before and after the invisible mark you set,

and observe the results. Killing and yanking text is one of the most complicated, confusing, and hard-to-explain features of Mince. However, once you do understand it, it's also one of the most useful and convenient features. Take your time and experiment enough to make sure that you understand these commands.

When you're all done type a C-X C-C and answer "Y" for "yes" when Mince asks you if you want to abandon the text buffer without saving it. You wouldn't want to destroy the file "lesson6.doc" for the next person, after all. We'll learn more about text buffers and neat tricks you can do with C-Y and lots of different buffers in Lesson 8.

Lesson 7: Text Processing Commands

This lesson is about commands which operate upon words, sentences, and paragraphs. You may remember that in Lesson 2 we made an analogy between the "C-" commands and the "M-" commands (although we didn't call them that back then) and thereby learned M-F, M-B, M-D, and M- to operate on words. We also learned M-A and M-E for moving by sentences and M-< and M-> for moving to the beginning and end of the entire buffer of text.

In this lesson you should have some text to play around with, and it should have a few paragraphs. The text of this lesson is an ideal example, but you may want to use some text that you have been working on and are familiar with. Read the file into Mince with C-X C-R. (If you decide you want to play with this lesson text, read in "lesson7.doc".)

The first three commands we will learn are M-U, M-L, and M-C. All three work on words. M-U stands for "Uppercase word". M-L stands for "Lowercase word". M-C stands for "Capitalize word". Try each of these on some words.

You may notice one very strange thing about these commands -- they do not actually look for a word in order to begin recasing letters. They merely look for the first alphabetic or numeric character starting at the cursor. For example, if you had the word "Macpherson" and positioned the cursor on the "p", a M-C would have produced "MacPherson", a M-L would have left the word unchanged, and a M-U would have created "MacPHERSON". In any case, the word casing commands would have left the cursor on the space following the word. Although these commands do not look backwards for the beginning of a word to work on if they are in the middle of one, they WILL look forward as much as necessary. (You may remember something similar from the discussion on M-F and M-D in Lesson 2.) Thus, C-U 5 M-U will uppercase the 5 following words, regardless of whether they are separated by spaces, newlines, punctuation, or other special characters.

The next command we will learn deals with sentences. It is M-K, the "Kill sentence" command. M-K is similar to C-K; it saves what it kills in case you want to do a C-Y later. Also, just as a C-K typed in the middle of a line only kills forward to the end of the line, a M-K, if typed in the middle of a sentence, will only kill from that point to the end of the sentence. (If

you are in the middle of a sentence and want to kill it all, beginning to end, type M-A (beginning of sentence, learned in Lesson 2) then M-K.)

M-K can be fooled by abbreviations because they have periods in them and hence look just like ends of sentences. But better too little deleted than too much. If M-K ever stops before you want it to, just type it again, and the "rest" of the sentence will disappear. Try a few M-K's and then a C-Y to restore it all. You may notice that the last M-K will not delete the two spaces after sentence punctuation. It truly only deletes from where the cursor is to the next end-of-sentence which follows it. You may have to clean up the extra spaces manually.

The rest of the commands we will learn about in this lesson deal with entire paragraphs of text. The two simplest commands are M-[and M-]. They move to the beginning and end, respectively, of whatever paragraph you are in. (You can remember these commands because they look very similar to M-< and M->, which we learned in Lesson 2.) If you are inbetween two paragraphs and not really "in" either one, M-[will move to the beginning of the preceding paragraph and M-] will move to the end of the following one.

Try these a few times, at various places in the text. You may notice the cursor stop in places you didn't think were paragraphs, for example in the middle of lists or in front of indented examples. So, what makes a paragraph? As far as these and all other Mince paragraph commands are concerned, a paragraph begins (or ends, if you prefer to think of it that way) with:

- (1) A blank line. (that is, two newline characters in a row.)
- (2) A line started by hitting the TAB key. (That is, a newline character followed by a tab character.) Note that this is NOT the same as a line with leading spaces.
- (3) A line started with a commercial at-sign ("@"). This is for compatibility with a particular set of "text-formatter" commands; chances are rare that you will type a line with an atsign character as its first character in ordinary text.

Another command for dealing with paragraphs of text is M-Q. M-Q "fills" entire paragraphs of text, rearranging words and lines so that the right-hand margin is consistent. (We couldn't make M-Q stand for anything mnemonic; you'll just have to remember it by rote.) This allows you to type paragraphs as carelessly as you like with regard to right margin, and at the end of a typed paragraph, with a single keystroke go back and tidy up the text. It also allows you to keep the text neat in the same way: When modifying a previously existing paragraph you may add or delete words with no regard for existing margins, because M-Q can fix them up when you're done.

Try filling a few paragraphs. (To make M-Q work on a particular paragraph, position the cursor anywhere in it, then type the M-Q.) If M-Q doesn't do anything to the text, then the paragraphs are already as well filled as they can be. Try inserting some extra text into a line and doing another M-Q. If you notice M-Q joining any paragraphs together, this is because they were not separated properly. M-Q uses the same paragraphing rules as M-[and M-] do. If you want to make certain of how much text you are about to fill with M-Q, you can check to see where the edges are by typing M-[, then M-], then (if you're satisfied that the boundaries are correct) M-Q.

Of course, no harm is done if M-Q joins two groups of text which you desired as separate paragraphs. You can easily position yourself to where you want the new paragraph to occur and insert the proper separator (either a blank line or a leading tab). Then just M-Q the second new paragraph. Similarly, to join and refill two paragraphs, merely delete the separator characters and use M-Q. This can be particularly useful for modifying memoranda, manuals, or legal documents, where text is frequently repositioned to change paragraph structure and coherence.

You may have wondered how M-Q knows where the right margin is supposed to be. There is a default margin column, which you can set yourself if you choose. The command to do this is "C-X F". That is, type a C-X, and then an "F". (This command stands for "Fill column".)

There are two ways to use C-X F. One is with a universal repeat. Type:

```
C-U 70 C-X F
```

and you will see a message in the echo line saying "Fill Column = 70". Try setting the fill column to something between 75 and 80 and do a M-Q. The other way to set the fill column is "by eye". Move the cursor to somewhere in the middle of a line, and type C-X F without an repeat count. You will see a new fill column setting appear in the echo line. Type a M-Q again and notice where the new right margin is. If you give C-X F an repeat with C-U, it will set the fill column to that number. If you do not, it will set the fill column to wherever the cursor is at the time.

Another useful command is "C-X .". (That is, a Control X, followed by a period.) This command sets the paragraph indentation column, and is used to make an entire paragraph be indented away from the left edge of the screen. This command is analogous to setting the left margin on a typewriter, but this margin is used only by the text-filling commands. Type:

```
C-U 10 C-X .
```


and you will see the message "Indent Column = 10" appear in the echo line. Type a M-Q and look at the results.

The "C-X ." command is useful for making narrower paragraphs, perhaps for example text or quotes. To make one, make the indent column larger and the fill column smaller, then type the text and fill it with M-Q. Then, return the fill and indent columns to their original positions. (The standard settings are usually gotten by "C-U 0 C-X ." and "C-U 65 C-X F".)

If you find typing M-Q repeatedly while inserting text annoying after a while, you may be interested in "Fill Mode". This causes carriage-returns to happen automatically while you are typing, so that not only do you not have to worry about the right margins, but you may never have to type M-Q unless you go back to repair old text. For more information look in the Mince User's Manual under "modes".

Lesson 8: Buffers

You may remember the word "buffer" from Lesson 3, when we learned how to read and write files from the text buffer. As you recall, we said that the buffer was a place for storing the text while we manipulated it, and that files were copied into it and replaced from it.

Mince has more than one buffer for manipulating text. This can be handy when you are working on several files at once. For example, this manual is made up of several chapters, and if we wanted to edit two or three of them, making changes to one based upon the others, it would be nice to be able to read them all into Mince at the same time. Each file we chose to use could be read into one of Mince's buffers.

For example, start using Mince to edit this chapter, called "lesson8.doc", by typing:

```
C-X C-R lesson8.doc <CR>
```

to Mince. (Remember that the disc drives are labelled, and that if the file can't be found, you may have to type "B:lesson8.doc" or some such for the file name.) (Remember also that "<CR>" stands for "carriage-return. If you should forget what character to terminate the filename with, just look in the message asking what file to read; it will always have the "<CR>" in it for file commands.)

Look at the mode line at the bottom of the screen. Notice the portion that says "main: lesson8.doc". In the mode line, "main" is the buffer name. Since we have multiple buffers for storing text, they must be named in some way, just like files or disc drives. Buffer names may be from one to eight characters long. "main" is the one you get automatically when you start Mince up. Each of these buffers of text also has a filename associated with it. In this case, the filename is "lesson8.doc". We talked about filenames in Lesson 3.

There is a command which will list on the screen what buffers of text currently exist. This command is C-X C-B, the "list Buffers" command. Try it now. Type:

```
C-X C-B
```

You should see a display at the top of the screen, overwriting whatever text was there before. The text is not gone, just momentarily not displayed; this is an exception to the rule that what you see on the screen is what is in your document. Take a quick look at the display line with the buffer list on it. It has the name, "main", and the file name, "lesson8.doc", and a number, which tells you how many characters are in the buffer of text. Type a C-L now to redisplay the screen. The buffer list has been replaced by the original text again.

There is a command to create a new buffer, the C-X B command. It stands for "goto Buffer", and looks similar to the C-X C-B command, so you can remember them both fairly easily. Try this command now. Type:

```
C-X B
```

Mince will ask you for the name of a buffer to use, by displaying a message in the echo line. Type in the name "other", followed by a carriage-return. Mince will ask you if you want to create a new text buffer. We do, so type a "Y" to answer yes. Now the screen is devoid of text. Look at the mode line, and notice that the buffer name is now "other". We moved into a brand new text buffer, which has no characters in it. Notice also that the filename in the mode line associated with this buffer is called "DELETE.ME" This is so that if you mistakenly type a C-X C-W command to write the file without giving a filename, the text will be stored in a conspicuously-named file. Type a few characters just so the buffer isn't empty.

Now type a C-X C-B again to see a new list of the text buffers. Notice that the list now shows two buffers, "main" and "other". Going to a new text buffer did not delete the old one; it is just waiting for whenever you want to go back to it. Do that now; type:

```
C-X B main <CR>
```

The Lesson 8 text will appear again. Note that C-X B didn't ask you if you wanted to create a new buffer, because one by that name already existed. Type the C-X C-B command again, to list the buffers. Notice that there is an asterisk beside the "other" buffer. This means the same thing as the star on the mode line does: the text buffer hasn't been written out to a file since it has been modified.

Now go to another new buffer, called "lesson6". Type:

```
C-X B lesson6 <CR> Y
```

(The "Y" is in answer to the question asking if we want to create a new buffer.) Now that you're in the "lesson6" buffer, read in the file containing Lesson 6. Type:

```
C-X C-R lesson6.doc <CR>
```

The text will appear (if you got the file name right) and the mode line will now have a section saying "lesson6: LESSON6.DOC". Type a C-X C-B again to get your bearings, if you like.

It is usually very useful to have the buffer name be the same as the first half of the file name which Mince and the computer system use. We just accomplished that by creating a buffer with a name appropriate to the file we were about to read in. But Mince can do this automatically for us. The command C-X C-F (for "Find File") will read in a file, in the same manner as C-X C-R, but will automatically create a buffer of the appropriate name for it. Try it now. Type:

```
C-X C-F lesson4.doc <CR>
```

The mode line will now say "lesson4: LESSON4.DOC", and the text of Lesson 4 should be on the screen.

The C-X C-F find-file command does just a little more than automatically selecting a "nice" buffer name. It will look through all the Mince buffers you have to see if the file you want to find has already been read into a text buffer before. If so, it just switches to that buffer, rather than creating a new one and reading in the file. This is almost certainly what you want; if you had made changes to a buffer containing a file and then did a find-file, you would want to see the modified version. Try it now. Type:

```
C-X C-F lesson6.doc <CR>
```

Note that this puts you back in the "lesson6" buffer. Try:

```
C-X C-F delete.me <CR>
```

This puts you into the buffer called "other", with its original file name. So, C-X C-F always does an automatic C-X B command for you, either to a buffer which contains the file name you want, or to a brand new buffer into which it reads the file. It effectively prevents you from ever having to remember whether or not you had read in a file. Just use C-X C-F all the time.

Do a C-X C-B to get a handle on what text buffers and files we have again. There is certainly a lot of junk, so let's get rid of some of those old buffers. The command to do this is C-X K (for "Kill buffer"). Type:

```
C-X K lesson4 <CR>
```

This will kill the buffer called "lesson4", which contains the file "lesson4.doc". Be sure to keep the two distinct in your mind. C-X K and C-X B work with buffer names, and C-X C-F, C-X C-R, and C-X C-W all work with file names. Do a C-X C-B again,

and notice that the buffer named "lesson4" is no longer there.

Now type:

C-X K other <CR>

This command is intended to delete the buffer called "other", which just happens to be the one on your screen now. Mince will not delete a buffer which we are currently working on, and so it asks us which buffer we would like to go to instead. Type "main" followed by a carriage- return, and Mince will then switch you back to the "main" buffer (which has the Lesson 8 text in it), and try to delete the "other" buffer.

But there is another message at the bottom of the screen: Mince will ask you if you are sure that you want to kill the buffer, because it has some text (those few characters) in it which has not been written out to a since the changes were made. (Remember the star in the C-X C-B listing and at the right end of the mode line now?) Answer "Y" for yes, and let Mince delete the buffer. You can check this out with a C-X C-B listing if you like.

What advantages does using several buffers have besides allowing you to look at several files back and forth? It allows you to move or copy text from one to the other as well. In Lesson 6, we used the killing mechanism (or C-W wiping or M-W copying mechanism) to move or copy text from one place to another in a file. This method works on multiple files in separate buffers as well. You can kill some text in one buffer, do a C-X B command to another buffer, and yank back the killed text in the new buffer.

Let's work through an entire example in detail. This is a chance for us to review some of the many commands that you have learned in Lessons 6, 7, and 8. The task is to take the first paragraph of this text, Lesson 8, and make it the first paragraph of Lesson 6. The buffer which we are now in, "main", contains the file "lesson8.doc". The other buffer, "lesson6", contains the file "lesson6.doc".

- (1) M-< to the beginning of lesson8.
- (2) C-N down to the first line of the paragraph.
- (3) C-@ to set the mark at the beginning of the first line.
- (4) M-] to get to the end of the paragraph.
- (5) C-W to wipe out the paragraph.
- (6) C-X B to the buffer "lesson6"

- (7) M-< to the beginning of that buffer.
- (8) C-N down to just before the start of the first paragraph.
- (9) C-Y to yank the text back. It should now be the first paragraph of Lesson 6.

Don't forget that the text is still yankable again; you may want to go back to the other buffer and repair it with a C-Y.

We have learned some commands in this lesson which you might not use quite at first. Sooner or later, though, you will be using them regularly. If you get into the habit of using C-X C-F to read in a file initially, you will find it easier to use the multiple buffers later when you need to. Experiment some more with buffers and moving text back and forth among them. When you exit Mince with the C-X C-C command, be sure NOT to write out the buffers, so that Lessons 6 and 8 are intact for the next person to use them.

Epilogue

This document is merely an introduction to the features of Mince. Further information on commands ranging from the obscure (e.g. C-T, a command to transpose characters) to the sublime (e.g. M-C-R, the Query Replace command, which replaces occurrences of one string with another, allowing you to view results, and either accept or reject them) is available in the Mince User's Manual. Other useful features are available, such as "fill mode" or "page mode", which allow certain commands to do special things when inserting or deleting text.

Having practiced all the commands used in these lessons, you should be able to perform almost all duties required of you. There are a few commands left to learn, however, which can make the job of editing a little easier. Read the user's manual when you get a chance! Best wishes!

You are looking at the Mince tutorial. While you can read this in hardcopy form, it is more helpful to sit down at a video terminal and use Mince to read it. Just type the command "mince prgintro.doc".

Mince commands are generally shifted by the CONTROL key (sometimes labelled CTRL or CTL) or prefixed by the ESCAPE key (sometimes labelled ESC or ALT). Rather than write out ESCAPE or CONTROL each time we want you to prefix a character, we'll use the following abbreviations:

C-<char> means hold the CONTROL key down and type a character.
M-<char> means type the ESCAPE key, release it, then type the character. (The "M" stands for "Meta-command"; ESC is a substitute for another type of shift key, the imaginary "meta-shift" key.)

Thus, C-F would be hold the control key and type F. You will often be asked to type characters to see how they work; don't actually do this, however, until you see >> at the left of the screen. For instance:

>> Now type C-V (View next screen) to move to the next screen. (go ahead, do it by depressing the control key and V together). From now on, you'll be expected to do this whenever you finish reading the screen.

Note that there is an overlap when going from screen to screen; this provides some continuity when moving through the file.

The first thing that you need to know is how to move around from place to place in the file. You already know how to move forward a screen, with C-V. To move backwards a screen, type M-V (depress and release the <ESC> key, then type V.)

>> Try typing M-V and then C-V to move back and forth a few times.

SUMMARY

The following commands are useful for viewing screenfuls:

C-V Move forward one screenful
M-V Move backward one screenful
C-L 'Refresh' the current screen.

>> Try C-L now. (You'll notice that it centers the screen where the cursor currently is.)

BASIC CURSOR CONTROL

Getting from screenful to screenful is useful, but how do you reposition yourself within a given screen to a specific place? There are several ways you can do this. One way (not the best, but the most basic) is to use the commands Previous, Backward, Forward and Next. As you can imagine, these commands (which are given to Mince as C-P, C-B, C-F, and C-N respectively) move the cursor from where it currently is to a new place in the given direction. Here in a more graphical form are the commands:

```

          Previous line, C-P
                :
                :
Backward, C-B .... Current cursor position .... Forward, C-F
                :
                :
          Next line, C-N

```

You'll probably find it easy to think of these by letter. P for previous, N for next, B for backward and F for forward. These are the basic cursor positioning commands and you'll be using them ALL the time so it would be of great benefit if you learn them now.

>> Try doing a few C-N's to bring the cursor down to this line. Move into the line with C-F's and up with C-P's. Now use these four commands to play around a little. Try moving off the top of this screen and see what happens.

When you go off the top or bottom of the screen, the text beyond the edge is shifted onto the screen so that your instructions can be carried out while keeping the cursor on the screen.

>> Try to C-B at the beginning of a line. Do a few more C-B's. Then do C-F's back to the end of the line and beyond.

If moving by characters is too slow, you can move by words. M-F (remember, type <ESC> F) moves forward a word and M-B moves back a word.

>> Type a few M-F's and M-B's. Intersperse them with C-F's and C-B's.

You will notice the parallel between C-F and C-B on the one hand, and M-F and M-B on the other hand. Very often Meta characters are used for operations related to English text whereas Control characters operate on the basic textual units that are independent of what you are editing (characters, lines, etc). There is a similar parallel between lines and sentences: C-A and C-E move to the beginning or end of a line, and M-A and M-E move to the beginning or end of a sentence.

>> Try a couple of C-A's, and then a couple of C-E's.
Try a couple of M-A's, and then a couple of M-E's.

See how repeated C-A's do nothing, but repeated M-A's keep moving farther. Do you think that this is right?

Here is a summary of simple moving operations including the word and sentence moving commands:

C-F	Move forward a character
C-B	Move backward a character
M-F	Move forward a word
M-B	Move backward a word
C-N	Move to next line
C-P	Move to previous line
C-A	Move to beginning of line
C-E	Move to end of line
M-A	Move back to beginning of sentence
M-E	Move forward to end of sentence
M-<	Go to beginning of file
M->	Go to end of file

>> Try all of these commands now a few times for practice. Since the last two will take you away from this screen, you can come back here with M-V's and C-V's.

These are the most often used commands. Like all other commands in Mince, they can be given arguments which cause them to be executed repeatedly. The way you give a command a repeat count is by typing C-U and then the digits before you type the command. (C-U stands for "Universal argument".) The digits are echoed at the bottom of the screen slowly, just after you type them. Notice that just after you type C-U, the message "Arg: 4" appears there. If no numbers are typed after the C-U, it executes the following command 4 times. For now, though, just type in numbers.

For instance, C-U 8 C-F moves forward eight characters.

>> Try giving a suitable argument to C-N or C-P to come as close as you can to this line in one jump.

INSERTING AND DELETING

If you want to type text, just do it. Characters which you can see, such as A, 7, *, etc. are taken by Mince as text and inserted immediately. You can delete the last character you typed by doing (sometimes labelled "DELETE" or "DEL" or even "RUBOUT"). More generally, will delete the character immediately before the current cursor position.

>> Do this now, type a few characters and then delete them by typing a few times. Don't worry about this file being changed; you won't affect the master tutorial, because this is just a copy of it in your Mince editing buffer.

Notice that a "*" appeared in the line at the bottom of the screen. This means that the text on your screen is different than the text you read in, and hasn't been written out to a file.

Remember that most Mince commands can be given a repeat count; Note that this includes characters which insert themselves.

>> Try that now -- type C-U 8 * and see what happens.

You've now learned the most basic way of typing something in Mince and correcting errors. You can delete by words or lines just as you can move by words or lines. Here are some of the delete operations:

	delete the character just before the cursor
C-D	delete the character that the cursor is positioned on
M-	delete the word before the cursor
M-D	delete the word after the cursor
C-K	delete (kill) from the cursor position to the end of line
M-K	delete (kill) to the end of the current sentence

Notice that and C-D vs M- and M-D extend the parallel started by C-F and M-F. C-K and C-E are similar to M-K and M-E.

Now suppose you delete something, and then you decide that you want to get it back? Well, whenever you delete something bigger than a character, Mince saves it for you. To yank it back, use C-Y. Note that you don't have to be in the same place to do C-Y; this is a good way to move text around. Generally, the commands that can destroy a lot of text will save it, while the ones that attack only one character, or nothing but blank lines and spaces, will not save them.

For instance, type C-N a couple times to position the cursor at some line on this screen.

>> Do this now, move the cursor and kill that line with C-K.

Note that a single C-K will kill the contents of the line, and a second C-K will delete the line itself, and make all the other lines move up. The text that has just disappeared is saved so that you can retrieve it. To retrieve the last killed text and put it where the cursor currently is, type C-Y.

>> Try it; type C-Y to yank the text back.

Think of C-Y as if you were yanking something back that someone took away from you. Notice that if you do several C-K's in a row the text that is killed is all saved together so that one C-Y will yank all of the lines. A way to tell if this is going to happen or not is the "+" which will appear on the line at the bottom of the screen. If it is present, whatever text is killed will be appended to whatever is already there.

>> Do this now, type C-K several times.

Now to retrieve that killed text:

>> Type C-Y. Then move the cursor down a few lines and type C-Y again. You now see how to copy some text.

FILES

In order to make the text you edit permanent, you must put it in a file. You put your editing in a file by writing or saving the file. If you look near the bottom of the screen you will see a line that starts with "Mince Version 2.3 (Normal) prgintro:" and continues with the filename PRGINTRO.DOC. This is the name of the permanent file in which the Mince tutorial is stored. This is the file you are now editing. Whatever file you edit, its name will appear in the same spot.

The commands for reading and saving files are unlike the other commands you have learned in that they consist of two control characters. They both start with the character Control-X. There is a whole series of commands that start with Control-X; many of them have to do with files, buffers, and related things, and all of them consist of Control-X followed by some other character.

C-X C-W	writes out the editing buffer
C-X C-R	reads a file into the editing buffer

In addition, each of these commands asks for a filename to use. Enter the name, and finish it by typing a carriage-return (<CR>).

>> Go ahead and try that now; type C-X C-W, and when Mince asks for a filename, type GARBAGE.TXT, then type a <CR>. Note that the mode line has now changed to reflect the new file name. (Don't forget to ERASE the file after you're done sometime.)

If you forget to write out your work and try to read another file, Mince will remind you that you made changes and ask you whether to clobber them. (If you don't save them, they will be thrown away. That might be what you want!) You should answer with a "N" to keep your editing buffer intact or a "Y" to clobber it and read in the new file anyway.

To make a new file, just C-X C-R it "as if" it already existed. Mince will echo "New File" at the bottom of the screen. Then start typing in the text. When you ask to write the file, Mince will really create the file with the text that you have inserted. From then on, you can consider yourself to be editing an already existing file.

Another command is available to prevent retyping filenames all the time.

C-X C-S saves the file

This command just rewrites the editing buffer to whatever file name is in the mode line at the bottom of the screen. It may save some typing.

It is not easy for you to try out making a file and continue with the tutorial. But you can always come back into the tutorial by starting it over and skipping forward. So, when you feel ready, you should try reading a file named "FOO", putting some text in it, and writing it; then exit from Mince and look at the file to be sure that it worked.

One more immediately useful command is C-X C-C, which tells Mince you'd like to stop editing. (Think of it as an augmented C-C, which usually works in the operating system to get you out of programs.) This does NOT save your file. It will ask if you really want to quit if you have not written out the editing buffer, however.

MODE LINE

If Mince sees that you have typed an <ESC> or C-U or C-Q or a C-X and have not typed the following character in the command sequence, it will show you the prefix you have typed in an area at the bottom of the screen. This line is called the "echo line"; it echoes numbers typed after a C-U, characters to be included in search strings, and some progress information when file I/O is going on. This is just the last line at the bottom.

The line immediately above this is called the MODE LINE. You may notice that it begins

```
Mince Version 2.3 (Normal) buffer: DRIVE:FILENAME -nn%- *
```

This is a very useful "information" line. You already know what the filename means -- it is the file you have read. What the -nn%- means is that nn percent of the file is above the cursor. The "*" means that the editing buffer has been changed since the file was last written. You also know what the "+" means in relation to the C-K command.

SEARCHING

Mince can do searches for strings (these are groups of contiguous characters or words) either forward through the file or backward through it. To search for the string means that you are trying to locate it somewhere in the file and have Mince show you where the occurrences of the string exist. The command to start a search is C-S. Down in the echo area, you will notice "Forward Search: <ESC>:" appear. Type in the string you want to search for (which will appear in the echo area also). When you finish, type the ESCAPE key and Mince will try to find the next occurrence of the string in your text.

If no such occurrence exists Mince beeps and tells you that the string was not found. In addition, if you decide you really don't want to search after all, type C-G and Mince will erase the search string and cancel the C-S command. (More generally, C-G cancels any command; for example, if you mistakenly type <ESC> or C-U when you didn't mean to, type C-G to flush the prefixes.) If you are in the middle of an search and type the DELETE key, you'll notice that the last character in the search string is erased.

>> Now type C-S to start a search. Type the word "search" followed by an <ESC>. Notice where the cursor is positioned to. Now type C-S again, immediately followed by <ESC>. Mince will search for whatever it searched for last time if no new string is given.

The C-S starts a search that looks for any occurrence of the search string AFTER the current cursor position. But what if you want to search for something earlier in the text? To do this one should type C-R for Reverse search. Everything that applies to C-S applies to C-R except that the direction of the search is reversed.

GETTING MORE HELP

In this tutorial we have tried to supply just enough information to get you started using Mince. There is so much available in Mince that it would be impossible to explain it all here. However, you may want to learn more about Mince since it has numerous desirable features that you don't know about yet. Documentation is available online and in hardcopy form. The Mince User's Manual completely describes the commands presented in this tutorial, as well as the more sophisticated commands, modes, and editing features.

CONCLUSION

You'll probably find that if you use Mince for a few days you won't be able to give it up. Initially it may give you trouble. But remember that this is the case with any editor, especially one that can do many things.

Mince User's Guide

January, 1981

This document Copyright (c) 1981 by Mark of the Unicorn

Table of Contents

1. Overview and Introduction	2
2. The Display	6
3. The Commands	9
4. Input/Output	15
5. Text Buffers	17
6. Mince Modes	20
7. Multiple Windows	23
8. Command Cross-Reference Index	25

Section 1: Overview and Introduction

1.1 What is Mince?

Mince is a display-screen oriented text editor, originally optimized for use on small computer systems. It is patterned after an editor called "Emacs", a text editor heretofore found generally at very large research computer installations.

1.2 Mince for Text Entry

Mince text entry has been made as easy as possible for the terminal operator. To enter text into a document, merely type it. No complicated commands are needed for this inherently simple operation. There are no "input" or "edit" modes, as in some text editors. Text which is entered is displayed as it is typed in. The screen display always shows text exactly as it will appear in the computer file and upon output; what you see is what you get.

1.3 Mince for Editing Text

Editing with Mince is easily and quickly learned. In general, Mince commands are mnemonically assigned to keys. For example, the editor commands which move forward in the text are associated with the "F" key and commands which move backward in the text are associated with the "B" key. Mince has an extremely powerful command set, but with the knowledge of only a small subset of those commands, most standard editing tasks can be accomplished with ease. Each key performs a command, and the result of the command happens immediately. There is no need to type carriage-returns to enter the editing commands. The screen display reflects the result of these commands as they are typed. This mnemonic command-naming and continuous interaction with the text rather than using a structured set of "editing requests" make Mince easy for the novice to learn yet efficient for experienced users to operate. The continuously-updated display helps all classes of users keep track of exactly what they are doing to the text.

1.4 Mince for Editing Programs

Programmers as a class appreciate superior speed and power from their editors. Mince supplies both. Several programs may be edited at once, and many of the textual editing features regarding sentences, words, and paragraphs are easily used with structured languages where such concepts translate to tokens,

statements, and blocks.

1.5 Mince and the Future

Mince is an editor which will not become obsolete with changes in hardware; it easily adapts to different terminal types without extra (and expensive!) software. Furthermore, Mince is written in a high-level language whose transportability will accompany the program development cycle for several years to come. The time spent learning Mince will not be wasted when 8080/Z80 systems have been replaced by newer hardware.

People who have not read either the Programmer's Introduction to Mince or the Mince Lessons should stop here rather than continuing with this user's guide. New users are best advised to take one of the introductory sets of lessons, sit down at a terminal, and get started. Mince is easiest to explain and learn by using it!

1.6 A Mince Glossary

The following symbols or terms are used in the command descriptions presented below. Familiarity with these items is a prerequisite for complete understanding of the Mince commands.

Point - This is the position in the text where editing occurs. All text which is typed into the document is entered at the Point and all text which is deleted is removed there as well. Each text buffer has its own Point. The Point is always BETWEEN any two characters. The cursor is always on the second of those two; hence the Point is just before the character that the cursor is on. Thus, many Mince commands do nothing but move the cursor to the right place on the screen (and thus move the Point to the right place in the text) to perform the next editing operation. It is perfectly easy to think of editing in terms of the cursor, but certain text entry and deletion operations can only be explained properly by thinking of this Point as between two characters.

Newline - This item is the character which causes Mince's display to move the cursor on the screen to the beginning of the next line. Newlines are always typed by hitting the carriage-return key on the terminal keyboard. Newlines are translated to the carriage-return/linefeed combination during text output, but they are treated as single characters for the purposes of text entry and deletion. During string input for commands which ask for string

arguments, the Newline will display as <NL>, in order to definitely indicate that a Newline is part of the string to be passed to the command.

move - This verb always means "move the Point" or "move the cursor". All Mince editing operations are done by "moving to" the appropriate place in a document (and on the screen) and making whatever textual changes are necessary there.

word - A word, as defined for the word movement and deletion commands, begins at the first alphanumeric character found and extends until a non-alphanumeric character.

sentence - A sentence, as defined for the sentence movement and deletion commands, begins at the first word found and extends until a punctuation mark is found.

paragraph - A paragraph, as defined for the paragraph movement or filling commands, extends until either a blank line (two Newline characters in a row), a line beginning with a tab (a Newline followed by a tab character), or a line beginning with an at-sign ("@") or period (".") (used by some text formatters to begin commands).

Mark - This is an invisible indicator in the text buffer. It may be set at a particular position by using the "Set Mark" command. Like the Point, it rests between two characters. Also like the Point, there is one Mark for each text buffer.

"C-" - This is the prefix which we will use in the text to refer to Control commands. These are formed by holding down the key marked "CONTROL", "CNTRL", "CTRL", or the like on the terminal keyboard while typing another character, usually alphabetic. Control characters are displayed on the screen as a caret or uparrow (^) followed by that character.

"M-" - This is the prefix which we will use in the text to refer to Meta-commands. These are formed by typing the key marked "ESCAPE", "ESC", "ALT", or "ALT MODE" on the terminal, then typing another character.

"M-C-" - This prefix is for the Meta-Control-commands. It is formed by typing the ESCAPE key, then typing the appropriate Control-command.

<CR> - A symbol used to refer to either the carriage-return key (labelled "RETURN", "CR", or "ENTER" on the keyboard). This key sends ASCII ^M, decimal 13.

<LF> - A symbol used to refer to either the linefeed key (labelled "LINEFEED" or "LF" on the keyboard, if it is

present at all). This key sends ASCII ^J, decimal 10.

 - A symbol used to refer to either the delete key (labelled "DELETE", "DEL", or "RUBOUT" on the keyboard). This key sends ASCII ^?, decimal 127.

<ESC> - A symbol used to refer to either the escape key (labelled "ESC", "ALT", "ESCAPE", or "ALTMODE" on the keyboard). This key sends ASCII ^[, decimal 27.

<TAB> - A symbol used to refer to the tab key. This key sends ASCII ^I, decimal 9.

<BS> - A symbol used to refer to the backspace key (sometimes labelled "BS" on the keyboard). This key sends ASCII ^H, decimal 8.

Section 2: The Display

The Mince screen display is divided into three areas. The major portion is the "window", where the text to be edited is displayed. Two smaller portions, the "mode line" and the "echo line", appear at the bottom of the screen, beneath the window.

2.1 The Window

The window occupies the upper portion of the screen. This area is where all text which is typed and all text which is about to be modified is displayed. The window displays about twenty or so consecutive lines of the document which is being edited. You may think of this as a window onto a larger entity of text, the entire document. The purpose of this window is to always show what the portion of the document contained in it looks like. Therefore, as text is inserted or deleted, the screen updates immediately. A fundamental principle of Mince is that what you see on the screen is what you actually have in your text.

2.1.1 The cursor

The screen display always has the terminal's cursor at some point in it, and as you will see, while the cursor is in the window, it is at the position where Mince commands will affect the text. The cursor is always positioned to the right of the Point in the document. (You may think of this as the Point being attached to the left edge of the cursor instead.) This property makes the cursor a very important object in the Mince display. Many Mince commands do nothing but move the cursor to the right place on the screen (and thus move the Point to the right place in the text) to perform the next editing operation.

2.1.2 Redisplaying the window

Since the window is always supposed to look exactly like its twenty-line portion of the text, any changes in the text must cause changes to the window display. Terminals have only one cursor, and screen updates are always performed at the cursor, since the text updates are also performed there. (Remember, the Point is attached to the cursor while the cursor is in the window.) Because of this, during screen update the cursor may move a great deal and there is no way to tell where the cursor is "supposed to be", (and thus where the Point actually is in the text) except by memory or dead reckoning. Fortunately, this update occurs very rapidly, and the cursor always returns in the window to where the Point is in the text.

It is possible to type very very quickly or to execute so many editor commands which have such large effects on the text that the screen update will not be able to keep up. In this case, Mince assumes that you know what you are doing and are not relying upon the screen redisplay. In these cases, it keeps trying to update the display to reflect the text in the buffer and the cursor's position in it, but it always processes typed-in characters before doing any other work. This priority can cause the screen to lag your typing somewhat. When you finally stop typing at full tilt, Mince will then update the screen to reflect the final state of the text in the window.

2.1.3 Special characters in the window

Special characters in the ASCII character set which are normally non-printing are displayed in Mince's window. This allows you to edit any type of file at all, with no worries about whether or not any information is missing from the screen displays. All the nonprinting characters can be represented by the ASCII control characters. The way Mince displays control characters on the display screen is by printing a caret or uparrow ("^") followed by the character which represents the control code in the ASCII collating sequence. (For example, ASCII Control E, decimal 5 (also known as ASCII "ENQ"), will be displayed as "^E".) All commands treat these as single characters, except the screen display which translates them for output. The Mince commands act on the "text" itself, not on the printed representation of it.

Mince displays the tab character (ASCII ^I, decimal 9) as moving the text which follows it on the line over to the next tab stop, rather than as a "^I". It displays a Newline as moving text which follows it to the beginning of the next line of the screen. Occasionally, Mince will display a Newline where there is not really one in the text. This is the only exception to the rule that what is on the screen is identical to what is in the text. If the text to be displayed would have run off the right-hand edge of the screen, Mince will display the rest of the line on the next line of the screen. It is usually easy enough to identify this condition, as the last word on the too-long line will frequently be cut in half.

2.2 The Mode Line

The second screen area of major importance is the line just below the text window. It is called the "mode line" and should look something like:

```
Mince Version 2.5 (Normal) buf1: X:FIRSTNAM.LST -23%- *+
```

This line tells you several things, namely:

(1) You are talking to Mince rather than the operating system.

- (2) You are typing in a mode called "Normal". In this mode, all commands typed are treated just as they are explained in the Mince Command List. There are other modes available, for example "Fill" mode or "Page" mode, each of which changes the Mince command set slightly. (See Section 6.)
- (3) You are editing text in a buffer named "buf1". This name is used when switching from buffer to buffer. (Buffers are explained in Section 5.)
- (4) You are editing a file called "X:FIRSTNAM.LST". This name is used when reading and writing files from the text buffer and the file system. (See Sections 4 and 5.)
- (5) The Point is approximately 23% of the way through the file. (Since the measure is only approximate, you may never see 100%, even if you are at the end of a file. For an exact measure of the cursor's position, try the "Where Am I" command, "C-X =".)
- (6) The buffer has been modified since it was last written out to the file. (This is indicated by the asterisk at the right edge of the mode line.)
- (7) The next text deletion command which stores deleted text in the kill buffer will append what it deletes to whatever is already contained in the "kill buffer" (see Section 5). (This is indicated by the plus sign at the right edge of the mode line.)

2.3 The Echo Line

The third important screen area is the one line of the screen left below the mode line. This area is called the echo line, for three reasons. First, it will echo any prefix characters (the first keystroke of any of the two-keystroke commands) typed. If any of these is typed and a particularly long amount of time (between .5 and 1 second) passes before the following character is typed, the prefix character will be indicated in the echo line. For example, if the Escape key is typed, the phrase "Meta:" will appear so that you know that you have indeed typed the prefix character. If not much time passes between the two keystrokes, this display is not performed. This behavior is designed to give confident users optimum response without the screen always flashing messages, yet give nervous users information on what they are doing. Secondly, the echo line is used for reading and displaying the arguments for some commands which require strings as input (e.g. the Forward String Search command). (See Section 3.4.2.) During these string-argument-read operations, the cursor will not be in the window in its usual spot attached to the Point. Finally, error messages from commands will appear at the far right edge of the echo line, as do informative messages which may be displayed while some commands are executing. Error messages displayed in the echo line are accompanied by a bell indicator, to alert the user of the error condition.

Section 3: The Commands

3.1 Text Insertion Commands

To Mince, everything you type is a command. Yes, everything. Even ordinary letters, numbers, and punctuation which you type are commands. They are very simple commands, though; they merely instruct Mince to insert themselves into the text. Because of this arrangement, Mince has no "insert mode" and "edit mode" which determine whether characters typed are commands or text to be inserted.

3.1.1 Ordinary text

All printing characters: a-z, A-Z, 0-9, space, and !"#\$\$%&'()*+,-./:;<=>?@[]^_{|}~` self-insert. The characters are inserted at the Point, that is, they are inserted just in front of where the cursor is on the screen. The Point is then left after the new character which was inserted. This definition means that on the screen, the cursor will move over one character position. Simply stated, text typed on a blank line moves the cursor just as a typewriter carriage would move after each character. Since characters are inserted at the Point, if the Point happens to be in the middle of some line of text, the rest of the line is moved over to make room for the new text.

3.1.2 The <TAB> key

This command, while nominally self-inserting (It inserts an ASCII ^I, decimal 9.) has a characteristic which affects the text screen display. When the tab character is displayed, the cursor is moved over to the next tab stop on the display screen before continuing to display text. This is not unusual, but is mentioned here because it is one of the characters which, when inserted in the text, takes up more than one column on the display screen.

3.1.3 The <CR> key

The carriage-return key inserts a character, the Newline, which causes the cursor to go the beginning of the next line on the screen. If a <CR> is typed when the Point is in the middle of a line, the line is split in two, with the portion of the line after the Point being moved down and turned into the next line. This is a little counterintuitive if you haven't used an editor before, but if you consider the Newline to be a character just like the other self-inserting ones, it makes

sense: <CR> inserts a Newline character at the Point, and moves the Point past the character. What this looks like to the operating system, when the file is stored in the file system, is a carriage-return/linefeed combination (ASCII ^ M/^J or decimal 13/10). In fact, this is what usually causes the display screen action of going to the beginning of the next line. But it's easier to consider the Newline character to be a single one, for text insertion and deletion purposes. (It is possible to enter just a carriage-return (^M) or a linefeed (^J) by means of the C-Q command, but typing the carriage-return key causes the Newline character to be entered into the text.)

3.1.4 The key

While this character is not an insertion command, it is so intimately tied to them that it must be mentioned here. The key is used for correcting typing mistakes; it deletes the last character typed. Since a Newline is treated as an ordinary character, a can delete <CR>'s typed just like any other character. A more complete description may be found in the Command List.

3.2 The Simplest Editing Commands

How are "true" editing commands differentiated from actual text? By the use of the terminal's CONTROL key. The most frequently used editing commands are one character long, and that character is a control character. For example, a C-F causes the cursor on a video display screen to be moved forward a character, and C-B causes it to be moved backward a character. The control characters are generated by holding down the control key (just like the Shift key) and typing the character. The case of a letter makes no difference to Mince command characters; "C-f" is the same as "C-F".

The commands are mnemonically named. Occasionally the mnemonics are stretched a little, but by and large the commands have some relation to the letter upon which they are placed. This makes it possible to remember in a natural manner all the commands which manipulate the text. The special function keys on the terminal keyboard are not usually used for another reason: it is not possible to use these keys without having the typist's hands leave the standard keyboard position. This choice means that it takes a little longer to learn the command set, but the time made up later by command touch-typing is well worth the effort.

In most editors, since there is a differentiation between text insertion mode and editing mode, it is easy to type the characters which make up the editor command set into the text. Since Mince commands are all control characters it is unlikely that you will want to type them into your text. If you do,

however, there is a command to "quote" the next character and insert it into the text directly rather than interpreting it as a command. (See C-Q in the Command List.)

3.3 The Two-Keystroke Editing Commands

As you will see, there is a progression in command names. The simplest and most-used commands are the easiest to type. Text itself, for example, is very easy to type. Those one-character commands which are textual characters self-insert into the document you are editing. Commands used to edit single characters or lines are quite frequently used, and are placed on the simple control characters. The more complicated and lesser-used commands are slightly harder to type and require two keystrokes.

3.3.1 Meta-commands

Some of two-character commands are called "meta-commands". These are generated by typing the ESCAPE key, then typing a character. Typing the escape key causes the following character to be treated as a Meta-command. If sufficient time elapses after typing the <ESC> and the following character, the message "Meta:" will appear in the echo area to indicate that Mince is waiting for a character to make up one of the Meta-commands. A C-G will cancel the Meta prefix and leave the text unchanged. Again, case makes no difference; the second character typed may be either upper or lower case. For example, typing ESCAPE F (M-F) moves the cursor forward a word, and ESCAPE B (M-B) moves backward a word. Commands which operate on words, paragraphs, or the like are usually two-keystroke meta-commands.

3.3.2 Meta-control-commands

There is another set of two-keystroke commands, which is a combination of the control commands and the meta-commands. These are the "meta-control-commands", which are given by typing an ESCAPE, then the appropriate control character.

3.3.3 Control X commands

Yet another (but the last!) set of two-keystroke commands is the set of "Control X commands." These are nothing more than an ordinary one-keystroke command prefixed by the one-keystroke C-X command. Similarly to the Meta prefix character (<ESC>), typing the C-X will display "Control X:" in the echo area while waiting for the second character in the sequence, and typing C-G will cancel the C-X prefix.

3.3.4 Relation between simple and two-keystroke commands

There is a relation between the control commands, the

meta-commands, and the meta-control-commands. Usually, the character upon which the commands is based has similar effects in both the control and meta-commands. For example, C-A moves to the beginning of a line and C-E moves to the end. M-A and M-E move to the beginning and end, respectively, of sentences. Occasionally, the control and meta-commands may be direct opposites instead. For example, C-V views the next screen of text, whereas M-V views the previous screenful. In almost all cases, though, they are related in some way.

3.4 Arguments to Commands

Most Mince commands are self-contained. Some of them, however, need parameters, or arguments, to accomplish their task. For example, the string search command needs to know what string to search for. The command which sets tab spacing needs to know what column increment to use. Or perhaps Mince wants to know if it should destroy a buffer of text to read in a new file. These arguments give the commands any extra information they need to perform properly. In particular, numeric arguments are useful, as they allow repetition of the other Mince commands.

3.4.1 Numeric arguments (repeat counts)

Commands may be given numeric parameters as arguments. These arguments usually are used as repeat counts. For example, an argument of 100 to the C-F command would cause the cursor to move forward 100 characters forward rather than the usual 1 character. Occasionally, arguments are used differently; for example, an argument to the command which sets the right margin specifies the column number to set it at, rather than the number of times to reset the right margin! Since all ordinary textual characters are actually commands too, they can be given arguments in order to insert several of themselves at once. For example, an argument of 20 to "*" will insert twenty stars into the text at the Point.

Of course, the manner of giving commands numeric arguments is itself a command! The C-U command specifies the numeric argument for the command which follows it. For example, if we wished to give the argument of 100 to the C-F command, we would type the C-U, then the number 100, then the C-F command. The number typed will be displayed in the echo line as it is being entered. Only positive numbers or zero may be entered; negative arguments do not exist.

3.4.2 String Arguments

Some commands will explicitly ask for character strings to be used as parameters for their execution. Examples of these are the string search commands and the file reading or writing commands. If such a string is needed a prompt will appear at

the bottom of the screen, in the echo area. The character string to be supplied to the command as an argument should be typed in.

Part of the echo-line display will be the prompt itself, which will describe what the string to be given to the command will be used for. Following this prompt will be a string specifying the termination character, either a <CR> or <ESC>. This character, when typed, will signal the end of the string argument and allow the command to process it.

If any typing mistakes occur while entering this string, the key will erase them, as it does in ordinary text. However, the rest of the editing commands will just insert themselves into the string argument, rather than editing it. This means that the command which usually quotes the command characters to insert them into the text (the C-Q command) is almost never necessary when entering a string argument. However, if the termination character is to be inserted into the string argument, it must be quoted.

The C-G command will abort execution of the command which is asking for the string argument. Naturally, it will thus terminate the string entry as well. If a C-G must be entered into the string argument, it too can be quoted as normal.

The <CR> key has a different effect than usual when typed as part of a string argument. It will display as "<NL>" rather than going to a new line. This both assures you that you have indeed inserted a Newline into your string (for example, to search for the end of a line followed by a word) and prevents the cursor from going off the screen.

Typing the termination character without typing any string first (called entering a "null string" in the Command List) causes the command to use an appropriate default for the character string. For example, the string search commands will use whatever string was last searched for, the file I/O commands will use whatever file name was last used in the text buffer, and the buffer commands will use whatever buffer name was last switched from.

3.4.3 Yes/No arguments

Some commands will ask Yes/No questions to get information. These questions, like the prompts for string arguments, will appear in the echo line, but they will not have any string termination character specified. Typing a "Y" or "y" or a space will answer yes to the question, and typing "N" or "n" or the key will answer no. Typing C-G will abort the command which asking the question; this is equivalent to the no answer to the question. The yes/no questions are usually asked if it is possible that some user command is about to destroy an entire buffer of text or make other large, irreversible

changes. For example, when leaving the editor, it will ask a yes/no question about whether or not the user really wants to exit if there is a text buffer which has been changed and not yet saved in the computer file system.

Section 4: Input/Output

4.1 Terminal I/O

Terminal input/output is fairly straightforward. Mince has several parameters which tell it how to do appropriate cursor positioning, character insertion, and line erasing on most standard terminals. The configuration program sets these up during your system installation. Character-at-a-time input is done from the keyboard; typeahead is automatically available, even during screen redisplay. Note that typeahead DOES NOT work during the other two forms of I/O. This is due to using the particular small computer (8080/Z80) and operating system which Mince is running on. Versions of Mince for slightly larger machines (e.g. PDP-11 or VAX) or other operating systems (e.g. UNIX or its look-alikes) do not have this restriction.

4.2 Text Buffer I/O

Text buffer I/O occurs because Mince implements a virtual memory system for storing text. Text files which are many times larger than available free memory space may be edited. Mince maintains a page-swapping file (called "MINCE.SWP") which it uses to page sections of text in and out of main memory on a least-recently-used basis. This paging affects the user in three ways:

- (1) Typing on the terminal keyboard is ignored during paging.
- (2) Screen redisplay may occasionally be interrupted for a short period if a page not in main memory is referenced.
- (3) If the keyboard remains idle for a certain length of time (settable in the configuration program) Mince will write pages which have been modified in main memory back to the disc swap file, in order to make swapping a little faster when the terminal keyboard is in use again. Typing at the keyboard causes this "housecleaning" operation to cease.

A warning message tells you that a text buffer I/O interruption is about to occur. The message "Swapping..." will appear in the error message area of the echo line.

It is possible to get the error message "Swap File Full" to appear on the mode line during file read operations, large text deletions, or during text entry (although the last is highly unlikely). This means that the total of all the text which Mince is storing is larger than the swapping file size, and that the file read or text entry operation did not

successfully complete because it was unable to find more room to create new pages for text storage. This condition is not fatal; editing may still be continued, although the operation which caused the swap file overflow almost certainly did not complete properly. There are two solutions for this condition: either remove some of the text buffers (via the "C-X K" command) which Mince is using and are no longer needed (if there is more than one) (you can, of course, write them out since the editor will still operate) or exit the editor and increase the swap file size using the configuration program. (It is, of course, possible to gain space in the swap file by deleting some of the text in one of the buffers rather than removing the buffer entirely, but this alternative is rarely used.)

4.3 File System I/O

File system I/O occurs only when it is requested. As with buffer I/O, terminal keyboard input is ignored during disc access. As this is not unexpected I/O, as is buffer page swapping, this limitation is not very important. File system names for the CP/M versions of Mince are of the following form:

X:FIRSTNAM.LST

"X:" is the optional disc drive name. "FIRSTNAM" is the one-through-eight character first component of the file name. ".LST" is the optional one-through-three character second component of the file name. (This second name must be preceded by a period if it is used.) It is probably wisest to use alphanumeric filenames, as different conventions are used when different operating system versions and programs try to parse these names later. When a file name is not specified to the file I/O commands, the file name which was associated with the text buffer in which the file read command was given (as shown at the right edge of the mode line) is used.

Section 5: Text Buffers

5.1 What is a Buffer?

Mince does not edit text directly on the files in the file system. Instead, it copies the text from whatever file you wish to edit into a "buffer" of text. Changes are made to this buffer, and are then saved back into the file if you ask for it, by giving the "write file" command.

5.2 Why Use Buffers?

This method has quite a few advantages. First, if you decide that the changes you have been making are not quite the right thing to do, you can exit the editor without having damaged your original file. Secondly, if the computer crashes while you are editing, your file will not be left open with incorrect information in the middle of it. Thirdly, you can read data from one file and write it out to a different one, leaving the original text file intact. Finally, you can have several buffers of text active at once while running Mince. This capability allows you to edit one document based upon information contained in another without constantly entering and exiting the editor. Additionally, with the use of the kill buffer (explained in Section 5.4 below), text may be moved from one document to another as well.

There are commands which allow you to switch from buffer to buffer, automatically create new buffers and select names for them while reading in new files. There is also a command to delete buffers, if necessary. Finally, there is a command to list all the buffers which currently exist, should you forget what files you have and have not read in to be edited.

5.3 What Other Information the Buffer Contains

In addition to the text in each buffer, several other pieces of information are stored with the buffer. A file name is associated with each buffer. It is the name of the file which has last been read from or written to while in this buffer. The default for this name (for example, if new text is typed into a buffer in which no read or write file commands have been issued) is "DELETE.ME". A Point and a Mark are also associated with each buffer. This means that when switching back and forth between buffers, the Mark and the Point in that buffer will not have moved, regardless of where you have moved any other buffers' Points or Marks, and that the display will

reappear in the same state it was when you left the buffer. Each buffer also has a "mode" associated with it. Usually this mode is "Normal" mode, but it can be changed. Modes are described in Section 6.

5.4 The Kill Buffer

There is one special buffer, called the "kill buffer". This buffer is used to save any deleted text which is deleted in clumps of more than one character at a time. For example, if you delete a line or two, they will be saved in this kill buffer. There is a command to retrieve what is saved in the kill buffer, so that if you make any large mistakes in deleting, you can undo what you have done. This feature is also used to move text from one place to another, as it acts as a temporary storage buffer.

The kill buffer stores only text deleted by commands which delete words, sentences, lines, or regions. The commands which delete characters are not considered "dangerous" enough to make mistakes with, nor good vehicles for deleting large quantities of text. Also, the kill buffer tries to store text in the proper order when deletions occur. Thus, deleting words backwards and deleting words forwards will append text to the correct end of the kill buffer, such that, if the text is yanked back out of the kill buffer again, all the words will be in the proper order.

The kill buffer will store and merge together consecutive text deletions. This means that if several line-kills are done in a row, they are all merged together. (You may have noticed that this deletion-merging was implied above by the discussion of word deletion order.) One kill buffer retrieval command will yank all the text which is in the kill buffer back at once. The kill buffer will only store one set of deletions at once, however. The first kill-buffer-saving text deletion which is done "opens up" the kill buffer. Thereafter, any successive text deletion commands given which save the deleted text in the kill buffer will merge it with whatever is already in the buffer. If any non-deletion commands are given thereafter, the kill buffer is "closed off". Following kill-buffer-saving text deletion commands will throw away whatever text is then in the kill buffer and open up a new set of killed text in the kill buffer. This behavior is reflected on the screen display. The plus-sign on the right side of the mode line indicates whether or not any text to be killed will be merged onto the kill buffer or not. If the plus is turned on, the text deleted will be added to what is already in the kill buffer. If it is off, the new deleted text will start a new kill buffer (and turn the plus-sign on again until a non-deletion command is given). There is a command which does nothing but "turn on the plus sign", so that text may, if desired, be deleted from several places and yet still merged together onto the kill buffer so that a single kill-buffer-retrieval command will yank it all

back.

The kill buffer is the mechanism used to copy or move text from one part of a document to another. A region of text is deleted (which saves it in the kill buffer). If it is to be copied, it is yanked back out of the kill buffer again, at the same spot, so that the text has not "really" been deleted. The copy of the deleted text is still stored in the kill buffer, however. Thus, to make another copy of it, the retrieval command can be used to yank it out again and again. Thus, it can be copied at any other place in the buffer, merely by issuing the command again. If the text is to be moved rather than copied, the text is merely not yanked back the first time at its original position, but only at the desired new position in the document. The kill buffer is constant across all buffers. Text which is deleted and saved in the kill buffer while editing one buffer can be yanked back into a different buffer. Thus, pieces of text can be moved from file to file by using multiple buffers, one to hold each file used in the move or copying operation, and by using the kill-buffer-saving text deletion commands and the buffer-retrieval yanking command.

Section 6: Mince Modes

Mince's "modes" are used to implement fundamentally differing strategies for interaction with text in the buffer. In general, if a large group of commands is to be changed in some consistent fashion or if entering text is to have a new meaning, the changes are combined into a mode.

6.1 Normal Mode

Mince begins execution in Normal Mode. This mode causes all ordinary textual characters typed to be inserted into the buffer at the Point. Other text in the buffer is shifted over to make room for the new characters entered. Normal Mode is the "starting point" for the Mince command set: Other modes may be added (on a per-buffer basis), but those modes only add functions or rebind commands to keys; unless the documentation for a mode specifically mentions Normal Mode commands which have been deleted, all commands which have not been assigned new functions are left with their Normal Mode meaning.

6.2 Fill Mode

Fill mode was designed to incrementally fill paragraphs. The auto-filling action makes text entry neater and prevents having to look at the display to keep track of margins. In Fill Mode, the space key's command checks to see if the previous word typed extends past the preset "fill column" (the same column used by the "Fill Paragraph" command), and if so, automatically inserts a Newline before the word. Then a space is entered into the text, as usual. This behavior means that typists need only hit the carriage-return key when they mean to enter blank lines or intentionally break text to begin on a new line. In order to insert a space past the fill column setting, the space command must be quoted (by using the C-Q command).

All other Normal Mode commands remain unchanged by Fill Mode. In particular, the Fill Paragraph command still works, and text entry and deletion is the same as usual. Thus, while the space command performs the auto-fill once, it does not keep the paragraph filled. Going back and inserting or deleting text can make the right margin more ragged again.

Note that this mode, like the Fill Paragraph command, does not justify the right margin by inserting spaces in the line. It leaves the right margin ragged, but fits as many words

as possible on a line, then automatically inserts a Newline.

6.3 Page Mode

Page Mode was implemented in Mince primarily as an aid for those who have had previous experience with other types of screen-oriented editors. In Page Mode, text insertion and movement commands treat the text buffer as if it were a two-dimensional grid of characters the width of the terminal screen (a "page" of text). This differs from Normal Mode, in which text is treated as a one-dimensional string of characters in which the Newline characters define lines of differing lengths.

6.3.1 Page Mode text entry

Unlike Normal Mode, text insertion while in Page Mode overwrites previously existing text. In order to actually insert characters without overwriting the character in the same position on the screen, the character must be quoted (via the C-Q command). Consistent with this, the <BS> key, rather than merely deleting the previous character, backs up and overwrites the character with a space. (Note that C-B does not overwrite, and actually removes the previous character from the line.) The carriage-return key inserts a Newline character and moves to the next line, exactly as it does in Normal Mode.

Frequently, you may desire to enter text in the middle of a line as in Normal Mode (i.e. insert rather than overwrite) for a short period of time without all the keystrokes involved in switching from Page Mode to Normal Mode and then back again or the overhead of quoting each character to be inserted. The easiest way to do this is to position the point in the buffer as you normally would, type the C-O command to split the right half of the line to a new one, type the text, then type a C-D command to delete the extra Newline and bring the right-hand part of the text back to the current line.

6.3.2 Page Mode line lengths

More differences from Normal Mode arise in the way Page Mode handles the end of a line and motion around Newline characters. In Page Mode, the horizontal character movement commands do not go past Newlines, as they do in Normal Mode. Instead, if the Forward Character command is given when the Point is just in front of a Newline, the line is extended by a character. On the screen, this means that the cursor will keep moving to the right on the same line, regardless of where the end of the text or the Newline was.

Similarly, the vertical line movement commands do not obey the same rules as in Normal Mode. In Normal Mode, if the Previous Line command (for example) is given at the end of a very long line and moves to a shorter line, the cursor will

move to the left on the screen, so as to be at the end of the shorter line. There was, after all, no text on the shorter line that far over. In Page Mode, however, the cursor will move directly up one line, not changing horizontal position. The short line would have been extended to the right in order to allow the cursor to rest in that position on the screen. This feature is very useful for adding or modifying columns of data as opposed to paragraphs of text.

Unfortunately, this uniform policy of extending lines whenever the cursor is positioned to a farther right column leads to some text storage inefficiencies. When writing the text buffer out to a file, any lines which were extended with spaces will not be trimmed back to the last nonblank character. Thus, as much of the page as the user has accessed by moving the Point will be stored when the text is written out. To counteract this waste, a command was included in Page Mode which deletes trailing whitespace on each line in the buffer (the C-X \ command).

These idiosyncracies are implemented in Page Mode to allow the user to consider the screen as a page of text, with screen-sized line boundaries and a consistent line length rather than lines which extend only as far as they were originally typed. Some other commands have been modified to give suitable effects for this mode. For example, the Beginning and End of Line commands have been changed to go to the first and last, respectively, nonblank characters on a line. See the Mince Command List for a description of the particular commands and their new properties.

Section 7: Two Windows

The Mince window system is a facility which augments the use of multiple buffers. By allowing the single screen window to be split into two separate windows, not only can many buffers of text be in use during one Mince session, but more than one of them may be displayed on the screen at once. This capability makes it easier than ever to modify one document based upon another or to move text from one buffer to another.

7.1 Creating a Second Window

The text window, which displays part of the document being edited, normally occupies almost all of the screen. When a second window is created (via the Two Windows command, "C-X 2"), each window occupies a little less than half of the screen. The windows are separated by a row of dashes the width of the screen.

It is important to distinguish between windows and buffers when using the system. A window may occupy all or part of the screen, and it merely shows a portion of what is in the text buffer currently being edited in it. The buffer stores the text, and the window displays the buffer. Thus, when the second window is initially created, since it displays the same buffer which was in the original (full-screen) window, the screen displays two copies of the same text, separated by a line of dashes.

7.2 Editing in a Window

Any editing requests affect the text buffer which is displayed in the window which has the cursor in it (the "current window"). This is just the same as when only a single window is displayed -- editing occurs at the cursor, because the cursor is attached to the Point in the buffer being displayed. Usually, different buffers are displayed in each of the windows, since a duplicate display is not too useful. (It is possible to display different parts of the same buffer in each window, however.) The buffer commands work for each window just as they did with the single window; the Select Buffer command (C-X B) may be used to show a different buffer in the current window. When switching from one window to the other (via the Other Window command, "C-X O"), the mode line will change to reflect the mode, file, and buffer names of whatever buffer is being displayed in the current window.

Using two windows does not affect the operation of the buffer system, since the Other Window command automatically switches to whatever buffer is displayed in the window. In particular, the window system does not affect the Kill Buffer. This makes it very easy to copy text (paragraphs, subroutines, whatever...) from one buffer to another. If the buffer into which the text is to be copied is displayed in one window, and the buffer from which text is to be taken is displayed in the other, there is immediate visual feedback on the successful completion of the Yank Killed Text command (C-Y).

7.3 Manipulating the Windows

If the number of lines displayed in one of the windows is too small, it can be increased by using the Grow Window command (C-X ^). Note that growing the current window shrinks the other window, however. A window may not be shrunk to display fewer than three lines; therefore, the other window can only be grown to a certain size.

Frequently, it is useful to scroll the text in the other window while making changes to the text in the current window. (Modifying source code based upon a compiler error listing file is a good example.) Going to the other window, scrolling it, and returning to the first window is rather tedious. Therefore, the View Next/Previous Screen Other Window commands (C-X C-V and C-X C-Z) allow you to view the next or previous screenful of text in the buffer displayed in the other window.

Finally, note that the two-window display is not permanent. The One Window command (C-X 1) causes the current window to become the only window and thus grow to occupy the entire display area again.

Command Cross-Reference Index

Add		
Mode		C-X M
Argument		
Numeric		C-U
Backward		
Character		C-B
Line		C-P
Page		M-V
Paragraph		M-[
Screen		M-V
Screen, Other Window		C-X C-Z
Sentence		M-A
Word		M-B
Beginning		
Buffer		M-<
Line		C-A
see also Backward		
Buffer		
Beginning		M-<
Delete, Kill		C-X K
End		M->
Go To		C-X B
List		C-X C-B
Capitalize		
Word		M-C
Center		
Line		M-S
Screen		C-L
Change		
see Buffer, Go To		
see also Delete		
see also Insert		
see also Replace		
see also Windows		
Character		
Backward		C-B
Delete		C-D,
Forward		C-F

Copy		
Region		M-W
see also Wipe		
see also Yank		
Delete		
Buffer		C-X K
Character		C-D,
Line (Kill)		C-K, M-C-K
Mode		C-X C-M
Region (Wipe)		C-W
Sentence		M-K
Whitespace		M-\
Word		M-D, M-
see also Yank		
Display		
see Screen		
see also Buffer		
see also Windows		
Down		
see Forward		
End		
Buffer		M->
Line		C-E
see also Forward		
Exit		C-X C-C
Files		
Find		C-X C-F
Read		C-X C-R
Save		C-X C-S
Write		C-X C-W
Fill		
Paragraph		M-Q
Find		
see Search		
see also File		
Forward		
Character		C-F
Line		C-N
Page		C-V
Paragraph		M-]
Screen		C-V
Screen, Other Window		C-X C-V
Sentence		M-E
Word		M-F

Go To

see Buffer
 see also Move
 see also Other Window

Indent

see Center Line
 see also Margins
 see also Tabs
 Newline and Auto-Indent C-J

Insert

see beginning of Commands List
 see also Quote

Kill

see Delete

Line

Backward, Previous	C-P
Beginning	C-A
Center	M-S
Delete, Kill	C-K, M-C-K
Delete Whitespace on	M-\
End	C-E
Forward, Next	C-N

Lowercase

Word	M-L
------	-----

Margins

Set Fill Column	C-X F
Set Indent Column	C-X .

Mark

Exchange Point and	C-X C-X
Set to Point	C-@ or M-<SPACE>
Whole Paragraph	M-H

Modes

Add	C-X M
Delete	C-X C-M

Move

see Backward
 see also Beginning
 see also Copy
 see also End
 see also Forward
 see also Kill
 see also Yank

Next

see Forward

Other		
Window, Go to		C-X O
Page		
see Screen		
Paragraph		
Backward		M-[
Fill		M-Q
Forward		M-]
Mark		M-H
Position		
see also Margins		
see also Tabs		
Previous		
see Backward		
Query		
see Replace		
Quit		C-X C-C
Quote		C-Q
Read		
File		C-X C-R
Redisplay		
see Screen		
Repeat		
see Argument		
Replace		
Query		M-C-R
String		M-R
Reverse		
see Search		
see also Transpose		
Save		
File		C-X C-S
Screen		
Backward, View Previous		M-V
Forward, View Next		C-V
Other Window, Next		C-X C-V
Other Window, Previous		C-X C-Z
Redisplay		C-L

Search		
Forward Search		C-S
Reverse Search		C-R
see also Replace		
Sentence		
Backward, Beginning		M-A
Delete, Kill		M-K
Forward, End		M-E
Set		
see Margins		
see also Mark		
see also Tabs		
Tabs		
Insert		<TAB> or C-I
Set		C-X <TAB>
Transpose		C-T
Undelete		
see Yank		
Universal		
see Argument		
Up		
see Backward		
Uppercase		
Word		M-U
Windows		
Grow		C-X ^
One		C-X 1
Other		C-X 0
Two		C-X 2
View Next Screen Other		C-X C-V
View Previous Screen Other		C-X C-Z
Wipe		
see Delete		
Word		
Backward		M-B
Capitalize		M-C
Delete		M-D, M-
Forward		M-F
Lowercase		M-L
Uppercase		M-U
Write		

Mark of the Unicorn

Mince User's Guide

File
Yank

C-X C-W
C-Y

The Mince Command List

All printing characters: a-z, A-Z, 0-9, space, and
!"#\$%&'()*+,-./:;<=>?@[]^_{|}~`

Self-insert

These characters are commands which insert themselves into the buffer. The characters are inserted at the Point, that is, they are inserted just in front of where the cursor is on the screen. The Point is then left after the new character. This means that on the screen, the cursor will move over one character position. Since characters are inserted at the Point, if the Point happens to be in the middle of some line of text, the rest of the line is moved over to make room for the new text. If any of these characters is given a numeric argument, that number of them is inserted into the buffer at the Point.

<LF> see C-J
(The linefeed key sends ASCII ^J, decimal 10.)

<CR> Newline Insert
This character is self-inserting as well, but causes the cursor to go the beginning of the next line on the screen. If a <CR> is typed when the Point is in the middle of a line, the line is split in two, with the portion of the line after the Point being moved down and turned into the next line. This is a little non-intuitive, but if you consider the Newline to be a character just like the other self-inserting ones, it makes sense. It inserts a Newline character at the Point, and moves the Point past the Newline character.

<TAB> Tab Insert
This character is also self-inserting (It inserts an ASCII ^I (decimal 9).), but causes the cursor to move over to the next tab stop. (To set tab increments for the screen display, see the "C-X <TAB>" command.)

 Delete Character Backward
Typing the delete key causes the last character typed to be removed from the text. Actually, what happens is that the character before the Point is deleted, so that the delete key, if used when the cursor is somewhere in a block of text, will always delete the character which is just to the left of the cursor. Since a Newline is treated as an ordinary character, typing a at the beginning of a line causes the current line and the previous line to be joined. A typed at the beginning of the buffer has no effect. This command does not save the deleted text

in the kill buffer.

<BS> Delete Character Backward

The backspace key is equivalent to the delete key, as described above.

<ESC> Meta-command Prefix

Typing the escape key causes the following character to be treated as a Meta-command. If sufficient time elapses after typing the <ESC> and the following character, the message "Meta:" will appear in the echo line to indicate that Mince is waiting for a character to make up one of the Meta-commands (or the Meta-Control-commands). The Meta-commands are explained in the second section below. A C-G will cancel the Meta prefix and leave the text unchanged.

C-@ Set Mark

This command sets the invisible Mark to the position where the Point is currently. On some terminals, You may have to type a Control-`<SPACE>` to get a C-@ command (it sends ASCII Null (`^@`), decimal 0). Others may be physically unable to generate this character. In that case, use the M-`<SPACE>` command instead. The message "Mark Set" is displayed in the echo line. (Remember this command because it sets the mark AT the Point.)

C-A Beginning of Line

This command moves the Point to just after the first Newline character preceding the Point. This has the effect of moving the cursor to the first character on the current line. Thus, repeated C-A's leave the cursor on the same line at the left edge of the screen. (Remember this command by either being at the beginning of the alphabet -> beginning of line, or by being at the left-hand edge of your keyboard -> left-hand edge of the screen.)

C-B Backward Character

This command moves the Point backward a character in the buffer. At the beginning of the buffer, C-B has no effect. Given a numeric argument, C-B moves back that many characters. Since Newline and `<TAB>` are treated as single characters, C-B skips over them as well. This means that a C-B issued at the beginning of a line will move the cursor to just after the last character on the previous line.

C-D Delete Character Forward

This command deletes the character which is after the Point. In other words, this deletes the character which the cursor is on. If the Point is before a Newline, the Newline is deleted; on the screen, if the cursor is after the last character on a line, a C-D causes the next line to be joined to the current line. Given a numeric argument, C-D deletes that many characters. A C-D typed at the end of the buffer has no effect. Note that the C-D command does not save the text it deletes in the kill buffer.

C-E End of Line

This command moves the Point to just before the first Newline character following the Point. This has the effect of moving the cursor to the last character on the current line. Thus, repeated C-E's leave the cursor on the same line at the right edge of the screen.

C-F Forward Character

This command moves the Point forward a character in the buffer. At the end of the buffer, C-F has no effect. With an argument, C-F moves forward that many characters. Since Newline and `<TAB>` are treated as single characters, C-F

skips over each of them as well. This means that a C-F issued at the end of a line will move the cursor to the first character on the next line.

C-G Abort/Cancel Prefix

This command aborts out of any other command which is requesting a string argument. It will also nullify any prefix characters typed to make Meta-commands or C-X commands. In other words, C-X C-G does nothing and M-C-G does nothing. C-G ignores any numeric argument it gets; therefore, this is a way to cancel numeric arguments as well. C-G also rings the terminal bell. (Think of this one as "beeping out" of command prefixes.)

C-H see

(This is the backspace key (ASCII backspace, decimal 8) on most terminals.)

C-I see <TAB>

(<TAB> sends ASCII ^I (decimal 9) on most terminals.)

C-J Newline and Indent

This command inserts a Newline character then inserts enough whitespace (tabs and spaces) to move the cursor horizontally so that the next character typed on the new line is in the same column as the first non-whitespace character on the previous line. This is frequently useful when writing programs in a block-structured computer language. It is functionally equivalent to a a <CR> and enough tabs and spaces to position the cursor properly.

C-K Kill Line

This command deletes text from the Point to the following Newline, unless the Point is just in front of a Newline, in which case it deletes the Newline itself. A positive argument to C-K repeats the command that many times; a zero argument to C-K deletes from the Point to the beginning of the current line. This means that a single C-K at the beginning of a line will "clear" that line, and typing it again will actually remove the Newline character and move all the others on the screen up a line. The C-K command stores text which it deletes in the kill buffer.

C-L Redisplay Screen

A C-L causes a Mince screen redisplay to occur. The current line will be centered on the screen (or placed on another screen line selected while running the configuration program). (Remember this one because ^L is the ASCII character for "form feed" (decimal 12), interpreted as page feed on printers and screen clear on some display terminals.)

C-M see <CR>

(The carriage-return key sends ASCII ^M, decimal 13.)

- C-N Next Line
This command moves the Point to the next line in the buffer. This has the effect of moving the cursor to the next line on the screen. It tries to maintain the same horizontal position as it had when vertical movement was begun, that is, C-N tries to move the cursor to the spot on the screen directly below it. If the line is too short, it moves the cursor to the rightmost position on that line (i.e., just before the Newline character). Repeated use of C-N (or C-P, the Previous Line vertical motion command) uses the original horizontal position. Given a numeric argument, C-N moves down that many lines. At the end of the buffer, C-N has no effect.
- C-O Open Line
This command inserts a Newline character but leaves the Point in front of the inserted character rather than after it, as is the case with <CR>. With a numeric argument, C-O inserts that many lines. This command is functionally equivalent to a <CR> followed by a C-B. It is primarily useful for splitting a line in half and inserting text immediately at the end of the first of the two resulting lines.
- C-P Previous Line
This command moves the Point to the previous line in the buffer. This has the effect of moving the cursor to the previous line on the screen. It tries to maintain the same horizontal position as it had when vertical movement was begun, that is, C-P tries to move the cursor to the spot on the screen directly above it. If the line is too short, it moves the cursor to the rightmost position on that line (i.e., just before the Newline character). Repeated use of C-P (or C-N, the Next Line vertical motion command) uses the original horizontal position. Given a numeric argument, C-P moves up that many lines. At the beginning of the buffer, C-P has no effect.
- C-Q Quote Next Character
This command is used to insert special characters into the text buffer which might otherwise be interpreted as Mince commands. It is also used to insert a string argument termination character into the string argument. If sufficient time elapses after typing the C-Q and the following character, the message "Quote:" will appear in the echo line to indicate that the C-Q command has been typed and is waiting for the next character. If C-Q is used while entering characters for a string argument to another command rather than while inserting text, the echo line is already in use and this message will not appear. The C-G command cannot abort out of a C-Q command; if typed, the C-G will be inserted into the buffer or string argument. Given a numeric argument, the C-Q command will

insert the following character into the buffer that many times, just as do the self-inserting (ordinary textual) characters.

C-R Reverse String Search

This command is very similar to the Forward String Search command, C-S, which is explained below. The string prompt message is "Reverse Search <ESC>:" instead, and the Point is left BEFORE the string if it is found between the current Point and the beginning of the buffer.

C-S Forward String Search

This command is used to search for particular strings of text. It displays the message "Forward Search <ESC>:" in the echo line of the screen display and awaits a string argument to search for as a string argument. If no string is entered (i.e., if the escape key is typed immediately after the C-S), whatever string was last given to a C-S or C-R string search command is used. Thus, repeated searches for the same string do not require retyping it each time. The string search is performed from the Point to the end of the buffer. If the string is found, the Point is left just after it. If not, the message "Not Found" is displayed in the error area of the echo line and the Point is left in its original position. The search is partially case-independent; that is, a lower case letter in the search string will match either a lower or an upper case letter in the buffer; however, an upper case letter will match only the upper case letter in the buffer. Given a numeric argument, the search is performed that many times. If the string is found that many times, the Point is left after that occurrence of the string. If not, the Point is not moved, and the "Not Found" message is displayed.

C-T Transpose Characters

This command transposes the characters before and after the Point (i.e., switches the character which the cursor is on with the one before it). It leaves the Point after the second one (i.e., moves the cursor to the character after the two just switched). Thus, successive C-T's will "drag" the previous character toward the end of the line. If the Point is at the end of a line, the two characters before the Point are transposed. This different behavior is useful when typing in new text; transposition typographical errors can be undone, since text insertion leaves the Point after the character just inserted. At the beginning of the buffer, the two characters following the Point are transposed since there is no character before the Point. Given a numeric argument, C-T will repeat whatever operation it would normally have done that many times. For example, a very large repeat count given to C-T in the middle of a line would first drag the character before the Point to the end of the line, then continuously transpose the last two characters on the line until the

repeat count was depleted.

C-U Universal Argument

This command is used to give numeric arguments to other commands. A number may be typed after typing C-U. If so, this number is the argument which is given to the next command typed. If not, the number 4 is automatically used. If sufficient time elapses after typing the C-U and any following character, the message "Arg: 4" will appear in the echo line to indicate that the C-U command may be given a typed number. If numbers are typed after the C-U, the "4" will be replaced with the number given. Note that will not remove the last digit typed to the C-U command; rather, the sequence of C-U followed by any number and a will cause repeated character deletion backwards. C-U's are multiplicative, that is, two C-U's typed in a row will cause the numeric argument given to the next command typed to be 16 rather than 4, and "C-U 7 C-U C-F" will move the Point forward $7 \times 4 = 28$ characters. If the C-U command is not used, the argument to any command will automatically be 1 instead. C-U may not be used to enter negative number arguments ("C-U -3" will enter "----3" into the text!) but may be used to enter zero as an argument (for example, "C-U 0 C-K").

C-V View Next Screen

This command moves the window so that it views the next screenful of text in the buffer. Incidental to this, the Point is moved down in the text as well, since the cursor always appears in the window display. There is an overlap of a few lines at the top and bottom of the screen so that repeated C-V's will have some continuity from screenful to screenful of text. Given a numeric argument, C-V will move the window down that many screens of text.

C-W Wipe Region

This command deletes ("wipes") the region of text between the Mark (see the C-@ command) and the Point. Wiped text is saved in the kill buffer.

C-X C-X Command Prefix

The next character typed is interpreted as one of the two-character C-X commands. See the list below. If sufficient time elapses after typing the C-X and any following character, the message "Control-X:" will appear in the echo line to indicate that the C-X command is waiting for another character to complete the command.

C-Y Yank Killed Text

This command inserts the contents of the kill buffer at the Point. It does not destroy the kill buffer, so that several C-Y's may be done to get several copies of the previously killed text. Given a numeric argument, C-Y yanks back the killed text that many times.

C-[see <ESC>
(The escape key generates the ASCII ^[(decimal 27) character.)

C-\ Delete Indentation
This command deletes the leading whitespace on the current line. It does not move the Point from its current position, however. This command does not save the whitespace it deletes in the kill buffer.

C-X <TAB> Set Tab Spacing

This command sets the tab increments for the display screen. These affect how far each tab character indents the following text. This command may be used in one of two ways: Given a numeric argument, it sets the tab spacing to that number. Given no argument, it uses the column number which the Point is at as the argument. Thus the tab spacing may be set "by eye".

C-X C-B List Buffers

This command lists all the buffers created in this editing session. Each element of the list has the following form:

```
buf1 * 1234 X:FIRSTNAM.LST
```

where "buf1" is the buffer name, the asterisk indicates that the buffer has not been written out since it has been modified, "1234" is the length of the buffer in characters, and "X:FIRSTNAM.LST" is the name of the file which was either last read from or written to in this buffer "buf1". As many of these lines as there are buffers are displayed at the top of the screen. (The kill buffer is not displayed in this list, since it cannot be used in the same manner as the others.) This command is useful if you forget which buffer a certain file has been read into. (Note its similarity to the "C-X B" command.) In order to remove the temporary display at the top of the screen, type any command (for example, C-L or C-G may be used).

C-X C-C Exit to Command Level

This command exits Mince, returning the user to the operating system. (CP/M users may remember it by thinking of it as an augmented C-C, the "standard" program interrupt character.) If any buffers have been modified since being written out, Mince asks the yes/no question "Abandon Modified Buffer(s)?" in the echo line.

C-X C-F Find File

This command puts you in a buffer editing a particular file, regardless of whether you have already read it into Mince or not. It displays the prompt "Find File <CR>:" in the echo line and waits for you to type in a file name as a string argument. If that file name is associated with any buffer available in Mince, it effectively does a "C-X B" command to display and allows you to edit that buffer. If there is no such file in a Mince buffer at the moment, Mince tries to create a buffer whose name is the same as the first component of the file name. If such a buffer is already in use, Mince displays the message "Buffer Exists!" and asks "Buffer to Use <CR>:" in the echo line and waits for you to supply a different name for the buffer. If you choose to re-use the buffer whose name is the first component of the file name (i.e. the one which Mince had originally chosen), just enter a carriage-return and the previous contents of the buffer will be lost. If

you choose to use a different buffer, enter its name. In any case, the filename given will be read in to the buffer selected, just as if the "C-X C-R" command had been given. (See its description for yet more ramifications.) In general, this command is functionally equivalent to a C-X B command, possibly followed by a C-X C-R command.

C-X C-I (same as C-X <TAB>)

C-X C-M Delete Mode

This command is used to delete modes from the current buffer's mode list. (Think of it as the inverse of the "C-X M" command.) It displays the message "Delete Mode <CR>:" in the echo line and waits for a mode name as a string argument. If no such mode exists, the message "Unknown Mode" is displayed at the right of the mode line. If the mode exists but has not been added to the buffer's mode list, this command does nothing. If it does exist and is on the buffer's list, it is removed and the mode line is updated to reflect the change in modes.

C-X C-R Read File

This command is used to read the contents of a file into the current buffer. It displays the message "Read File <CR>:" in the echo line and waits for the user to supply the name of the file to be read as a string argument. If a null string is entered, the file name currently associated with the buffer (i.e., whatever file name was last used in a file read or write command or "DELETE.ME" if there had been none) is used. If no such file is found on disc the message "New File" appears in the echo line, and the buffer is made empty. Since the Read File command always overwrites the current contents of the buffer, it checks to see if it has been previously modified without being written out. If it has, the Read File command queries the user with the yes/no question "Clobber modified buffer?" in the echo line. The Mark and the Point are set to the beginning of the text buffer when a file is read in.

C-X C-S Save File

This command is equivalent to typing "C-X C-W <CR>". See the Write File command description below.

C-X C-V View Next Screen Other Window

This command is used to scroll forward (or "down") the text in the window which the cursor is not currently in. Thus, it is functionally equivalent to the sequence "C-X O C-V C-X O". Given a numeric argument, the text is scrolled down that many times. (The actual number of lines passed depends upon the size of the other window.) If there is only one window being displayed, this command has no effect.

C-X C-W Write File

This command is used to write the contents of the current buffer to a disc file. It displays the message "File to Write <CR>:" in the echo line and waits for the user to supply a file name as a string argument. If the null string is entered, the file name currently associated with the buffer (i.e., whatever file name was last used in a file read or write command or "DELETE.ME" if there had been none) is used. If no file of the name given exists, one is created. If one exists, it is overwritten with the contents of the buffer.

C-X C-X Exchange Point and Mark

This command switches the Point and the invisible Mark in the current buffer. It is useful for determining the edges of a region which is about to be wiped with the C-W command. (Remember this command as standing for "eXchange".)

C-X C-Z View Previous Screen Other Window

This command is used to scroll backward (or "up") the text in the window which the cursor is not currently in. Thus, it is functionally equivalent to the sequence "C-X O M-V C-X O". Given a numeric argument, the text is scrolled up that many times. (The actual number of lines passed depends upon the size of the other window.) If there is only one window being displayed, this command has no effect.

C-X . Set Indent Column

This command is used to set the number of spaces which should be left blank at the left margin for the M-Q command, the M-S command, or while using Fill mode. Given a numeric argument, it sets the indent column to that number. If not given an argument, it sets the indent column to whatever column the Point is at in the text. The indent column is the first column in which text may appear when filling paragraphs or centering lines. (The fill column is the first column in which text may not appear.) Note that columns are numbered from zero. A default setting for the indent column (usually zero) is selected when the configuration program is run. This command displays "Indent Column is n" in the echo line.

C-X 1 One Window

This command makes whatever window in which editing is currently being done the only window on the screen. Effectively, this undoes the effect of the "C-X 2" command. If two windows are not being displayed on the screen, this command has no effect.

C-X 2 Two Windows

This command splits the display screen in half and turns the one window into two. This allows a buffer to be displayed in either the upper or lower window area (or

both). The window boundary is indicated by a line of dashes across the screen, separating the upper from the lower window. Initially, the second window will display the same buffer as the first window; that is, both windows will display the same buffer which was being edited when the Two Windows command was given. The cursor will be left in the upper window, and editing may be continued there. If there are already two windows on the screen, this command will have no effect.

C-X = Where Am I

This command displays (in the echo line) the location of the Point and Mark measured in characters from the beginning of the buffer and the size of the buffer measured in characters. It also displays the the Point's current horizontal position (column number).

C-X B Select Buffer

This command displays "Switch to Buffer <CR>:" in the echo line and waits for the user to supply in a buffer name to go to as a string argument. If the user picks a buffer name which is already in use, that buffer becomes the current buffer, it is displayed on the screen, and all subsequent editing operations are performed on it. If the user picks a new buffer name, Mince asks the yes/no question "Create New Buffer?". A no answer aborts the command execution, and a yes answer creates a new empty buffer. If no buffer name is supplied (i.e., the user enters a null string) whatever buffer was last switched from is switched to. Thus, repeated executions of the "C-X B" command cause the editor to switch back and forth between two buffers (once the "other" buffer is first set by actually giving C-X B a buffer name or by using the C-X C-F command, which automatically performs a C-X B).

C-X F Set Fill Column

This command is used to set the right margin used for filling text during the operation of the M-Q command or while using Fill mode or the line centering command, M-S. Given a numeric argument, it sets the fill column to that number. If not given an argument, it sets the fill column to whatever horizontal column the Point is at in the text. The fill column is the first column in which text may not appear when filling paragraphs or centering lines. (The indent column is the first column in which text may appear when filling paragraphs or centering lines.) Note that columns are numbered from zero. A default setting for the fill column (usually 65) is selected when the configuration program is run. This command displays "Fill Column is n" in the echo line.

C-X K Kill Buffer

This command is used to remove buffers from the Mince working set. It may be used to free up space in the swap

file if a "Swap File Full" error is encountered or to remove some of the visual (and mental) clutter which occurs with many buffers to keep track of. It displays "Delete Buffer <CR>:" in the echo line and waits for the user to supply the name of the buffer to be deleted as a string argument. If the null string is entered, the buffer name used is the last one switched from, as with the Select Buffer command, C-X B. If the buffer selected to be killed is the current buffer, Mince displays "Switch To Buffer <CR>:" in the echo line and waits for a new buffer name string argument. (Mince will not let you switch to the buffer you are about to delete, and displays an error message if you try to.) After performing the buffer switch (which is identical in function to the "C-X B" command), if any, Mince tries to delete the requested buffer. If it has not been written out since being modified, Mince asks the yes/no question "Delete Modified Buffer?" in the echo line.

C-X M Add Mode

This command is used to add modes to the current buffer's mode list. It displays the message "Mode Name <CR>:" in the echo line and waits for a mode name as a string argument. If there is no such mode, the error message "Unknown Mode" is displayed. Adding a mode to buffer in which it is already on the buffer's mode list has no effect.

C-X O Other Window

When two windows are displayed on the screen, this command switches from one window to the other. It automatically selects the buffer which is being displayed in the window being switched to. If only one window is being displayed on the screen, this command has no effect.

C-X ^ Grow Window

This command increases the number of lines used to display the window in which editing is currently being done. (Thus it decreases the number of screen lines available to display the other window.) Given a numeric argument, this command enlarges the window by that many lines. Windows cannot be smaller than three lines, thus a window cannot be grown such that it will force the other one to be smaller than that limit. If this command is given when only one window is being displayed on the screen, it has no effect. (Remember this command by thinking of the arrow "pushing" the window boundary up or down.)

M- Delete Word Backward

This command deletes text backward until it finds the beginning of a word. This means that if the Point is in the middle of a word, the first part of the word will be deleted. If the Point is at the end of a word, the entire word will be deleted. If the Point is after a word, all the intervening characters between the Point and the last character of the word will be deleted as well. Deleted text is saved in the kill buffer. Given a numeric argument, M- deletes that many words.

M-<SPACE> see C-@

(This command is implemented for those terminals physically unable to generate the character for the C-@ command.)

M-< Beginning of Buffer

This command moves the Point back to the beginning of the text buffer. Before doing this, it sets the Mark to the place where the Point currently is. This allows a C-X C-X command to get you back to where you started. (Remember this one because the less-than symbol Points in the direction you want to move.)

M-> End of Buffer

This command moves the Point to the end of the buffer. Before doing this, it sets the invisible Mark to the place where the Point currently is. This allows the C-X C-X command to get you back to where you started. (Remember this one because the greater-than symbol Points in the direction you want to move.)

M-A Backward Sentence

This command moves the Point to just before the sentence which it is currently in. If the Point is not in some sentence, it will be moved to the beginning of the previous sentence. Given a numeric argument, this command will move the Point backward that many sentences.

M-B Backward Word

This command moves the Point backward to just before the word which it is currently in. If the Point is not in some word, it will be moved to the beginning of the previous word. Given a numeric argument, this command will move backward that many words. Like all word commands, this one will skip over any intervening whitespace or punctuation while looking for the beginning of a word.

M-C Capitalize Word

This command capitalizes the current word. If the Point is in the middle of a word rather than in front of it, the letter which the cursor is on will be capitalized rather than the first letter of the word. Note that the numbers 0

through 9 are considered to be parts of words and have no capital form. The rest of the word is lowercased. This command leaves the Point just after the word which has been capitalized. Therefore, given a numeric argument, this command will capitalize and move forward past that many words. Like all word commands, this one will skip over any intervening whitespace or punctuation while looking for the beginning of a word.

M-D Delete Word Forward

This command deletes text until it finds the end of a word. This means that if the Point is in the middle of a word, the rest of the word will be deleted. If the Point is at the beginning of a word, the entire word will be deleted. If the Point is before a word, all the intervening characters before the word will be deleted as well. Deleted text is saved in the kill buffer. Given a numeric argument, M-D deletes that many words.

M-E Forward Sentence

This command moves the Point to the end of the sentence which it is currently in. If the Point is not in some sentence, it will be moved to the end of the following sentence. Given a numeric argument, this command will move the Point forward that many sentences.

M-F Forward Word

This command moves the Point forward to just after the word which it is currently in. If the Point is not in some word, it will be moved to the end of the next word. Given a numeric argument, this command will move forward that many words. Like all word commands, this one will skip over any intervening whitespace or punctuation while looking for the end of a word.

M-H Mark Whole Paragraph

This command Marks the paragraph which the Point is in (or which the Point is just before). It moves the Point to the beginning of the paragraph and sets the Mark to the end. This command is functionally equivalent to the sequence M-], C-@, then M-|. It is used for convenience with C-W and C-Y for copying or moving paragraphs. (Remember it by the "H" standing for "wHole".)

M-K Kill Sentence Forward

This command deletes text until it finds the end of a sentence. This means that if the Point is in the middle of a sentence, the rest of the sentence will be deleted. If the Point is at the beginning of a sentence, the entire sentence will be deleted. If the Point is before a sentence, all the intervening characters before the sentence will be deleted as well. Deleted text is saved in the kill buffer. Given a numeric argument, M-K deletes that many sentences. If the argument is zero, M-K deletes

backward from the Point to the beginning of the sentence.

M-L Lowercase Word

This command lowercases the current word. If the Point is in the middle of a word rather than in front of it, the letter which the cursor is on will be the first to be changed from upper to lower case rather than the first letter of the word. Note that the numbers 0 through 9 are considered to be parts of words and have no special lowercase form. This command leaves the Point just after the word which has been recased. Therefore, given a numeric argument, this command will lowercase and move forward past that many words. Like all word commands, this one will skip over any intervening whitespace or punctuation while looking for the beginning of a word.

M-Q Fill Paragraph

This command fills text in the current paragraph so that each line of text does not go past the right margin (set by the Set Fill Column command, "C-X F"), and so that each line after the first begins at the left margin (set by the Set Indent Column command, "C-X ."). Given a numeric argument, it sets the fill column to that number (i.e., executes an automatic "C-X F") and then performs the text filling operation. The text filling operation will move words between lines and insert or delete as many lines as are necessary to properly format the text.

M-R Replace String

This command replaces strings from the Point to the end of the buffer. It is functionally equivalent to the Query Replace (M-C-R) command, with the "!" option (to replace all following occurrences) specified. See the M-C-R command description for an explanation of the string arguments.

M-S Center Line

This command centers the current line, if possible, between the left margin (set by the Set Indent Column command, "C-X .") and the right margin (set by the Set Fill Column command, "C-X F"). If this is not possible, the line to be centered is left flush at the left edge of the screen. Given a numeric argument, it sets the fill column to that number (i.e., executes an automatic "C-X F") and then performs the centering operation. (Remember this command by the S-sound in "center".)

M-T Transpose Words

This command transposes the words before and after the Point. It leaves the Point after the second one. Thus, successive M-T's will "drag" the previous word toward the end of the line. If the Point is at the end of a line, the last word on the line is exchanged with the first word on the next line. If the Point is at the end of the buffer,

this command has no effect. If the Point is in the middle of a word, the two halves of the word are switched instead. Given a numeric argument, the word-switching is performed that many times.

M-U Uppercase Word

This command uppercases the current word. If the Point is in the middle of a word rather than in front of it, the letter which the cursor is on will be the first to be changed to upper case rather than the first letter of the word. Note that the numbers 0 through 9 are considered to be parts of words and have no special uppercase form. This command leaves the Point just after the word which has been recased. Therefore, given a numeric argument, this command will uppercase and move forward past that many words. Like all word commands, this one will skip over any intervening whitespace or punctuation while looking for the beginning of a word.

M-V View Previous Screen

This command moves the window so that it views the previous screenful of text in the buffer. Incidental to this, the cursor is moved backward in the text as well, since it must always appear in the window display. There is an overlap of a few lines at the top and bottom of the screen so that repeated M-V's will have some continuity from screenful to screenful of text. Given a numeric argument, M-V will move the window down that many screens of text.

M-W Copy Region

This command copies the region of text between the Mark (see the C-@ command) and the Point onto the kill buffer. (Remember this command by its relationship to the kill buffer and the C-W command.)

M-[Backward Paragraph

This command moves backward to the beginning of the current paragraph. If this command is given while the Point is not inside any paragraph, the Point will be moved to the beginning of the paragraph before the Point. Given a numeric argument, M-[will move the Point backward that many paragraphs.

M-\ Delete Surrounding Whitespace

This command deletes all spaces and tab characters on both sides of the Point. The whitespace deleted is not saved in the kill buffer.

M-] Forward Paragraph

This command moves forward to the end of the current paragraph. If this command is given while the Point is not inside any paragraph, the Point will be moved to the end of the paragraph after the Point. Given a numeric

Mark of the Unicorn

Mince Command List

argument, M-] will move the Point forward that many paragraphs.

M-C-H see M-

M-C-K Kill Entire Line

This command deletes the entire current line. It is similar to the C-K command, except that it will kill any text from the beginning of the line to the Point as well, and will kill the Newline after killing the text on the line. The killed text is saved in the kill buffer.

M-C-R Query Replace String

This command does string replacement, asking for some sort of confirmation at each occurrence of the string to be replaced. The search/replace operation is performed starting at the Point, toward the end of the buffer. Old and new strings (the string to be repeatedly searched for and the string to replace it with) are requested as string arguments with the prompts "Query Replace <ESC>:" and "With <ESC>:". At each occurrence of the old string, the user has the following command choices:

C-G Abort

This character will cause the replacing operation to stop. The Point will be left where it was in the middle of the searching operation (i.e., just after the current occurrence of the old string).

! Replace Rest

This character will replace all the remaining occurrences of the old string with the new one, without stopping at each to ask.

. Exit

This character will cause the query-replace to stop without searching for any remaining occurrences of the old string. The Point will be left at the position where the query-replace command was given.

, Replace and Request Confirmation

This character will cause the replacement to occur, then ask a Yes/No question to determine whether or not to actually leave the old string replaced by the new. Thus, the user can see the results of a replacement and decide whether or not to keep it.

Y (or y or space) Replace and Find Next

This character causes the current occurrence of the old string to be replaced with the new and the next occurrence of the old string in the buffer to be found.

<anything else> Don't Replace, and Find Next

Any other character typed will leave the current occurrence of the old string unchanged and search for the next one.

M-C-W Append to Kill Buffer

This command causes the next group of text deletion commands to append to the kill buffer if they otherwise would have started a new kill group. (This is effectively "turning on the plus sign" on the Mince screen display.)

(Remember this command by its relationship with the C-W command.)

Fill Mode Command List

space Auto Fill Space

This command inserts a space into the buffer and moves past the space. It also checks to see if the word just prior to the space extends past the right margin (set by the Set Fill Column command, C-X F), and if so, inserts a newline and inserts whitespace to the left margin (set by the Set Indent Column command, C-X .) before that word. Thus, a typist can type whole paragraphs without looking at the screen.

Page Mode Command List

All printing characters: a-z, A-Z, 0-9, space, and
!"#\$%&'()*+,-./:;<=>?@[]^_{|}~`

Self-overwrite

These characters are commands which overwrite the character at the Point with themselves. These characters will not overwrite a Newline character (instead, they extend the line), and if overwriting a tab character, will insert sufficient spaces to maintain the column position of the text following the tab.

C-A To First Non-White

This command moves the Point to just before the first non-whitespace character (i.e., non-space, non-tab) on the current line. This is similar to the Normal Mode C-A command, except that after moving back to just after the first preceding Newline, it moves forward until it encounters a non-white character in the line.

C-B Backward Character on Line

This command moves the Point backward a character in the buffer. It is similar to the Normal Mode C-B command, except that it will not skip over Newline characters (it always stays on the current line) and it treats tab characters as multiple spaces while moving rather than as single characters.

C-E To Last Non-White

This command moves the Point to just after the last non-whitespace character on the current line. This is similar to the Normal Mode C-E command, except that after moving forward to the first following Newline, it moves backward until it encounters a non-white character in the line.

C-F Forward Character on Line

This command moves the Point forward a character in the buffer. It is similar to the Normal Mode C-F command, except that it will not skip over Newline characters (it always stays on the current line) and that it treats tab characters as multiple spaces while moving rather than as single characters. Thus, moving to a Newline and typing C-F will extend the line by one more character to the right.

C-H Overwrite Character Backward

This command moves backward a character in the buffer and overwrites it with a space, leaving the Point before the

overwritten space. It is functionally equivalent to typing C-B <SPACE> C-B. (C-H is usually available by typing the <BS> key.)

C-N Next Line Forced

This command moves the Point to the next line in the buffer. It is similar to the Normal Mode C-N command, except that rather than trying to preserve the horizontal column position when moving from line to line, it forces the horizontal column position to be the same. Thus, if the line to be moved to has fewer columns than the current column position, it is extended the appropriate amount with whitespace before the cursor is moved. This has the effect of always moving the cursor directly vertically, rather than "hugging" the right margin of the text, as may occur in Normal Mode.

C-P Previous Line Forced

This command moves the Point to the previous line in the buffer. It is similar to the Normal Mode C-P command in all respects except those mentioned above for the C-N command.

C-Q Quote Next

This command "quotes" the next character typed; that is, it inserts the character literally into the buffer rather than performing the command associated with it. This command is identical in operation to the Normal Mode C-Q command. The only reason it is mentioned here is that when ordinary textual characters are quoted with C-Q, they are inserted into the buffer as in Normal Mode, rather than overwritten on the buffer as in Page Mode.

C-X \ Delete Trailing Whitespace

This command examines every line in the buffer and deletes trailing blanks from each line. This is useful to delete any trailing blanks which may have been created by the screen cursor positioning commands in Page Mode. (Remember this command by its similarity to M-\, the Delete Surrounding Whitespace command.)

The Mince Commands

All printing characters: a-z, A-Z, 0-9, space, and
!"#\$%&'()*+,-./:;<=>?@[]^_`{|}~`

Self-insert

<LF> see C-J

<CR> Newline Insert

<TAB> Tab Insert

 Delete Character Backward

<ESC> Meta-command Prefix

C-@ Set Mark

C-A Beginning of Line

C-B Backward Character

C-D Delete Character Forward

C-E End of Line

C-F Forward Character

C-G Abort/Cancel Prefix

C-H see

C-I see <TAB>

C-J Newline and Indent

C-K Kill Line

C-L Redisplay Screen

C-M see <CR>

C-N Next Line

C-O Open Line

C-P Previous Line

C-Q Quote Next Character

C-R Reverse String Search

C-S Forward String Search

C-T Transpose Characters
C-U Universal Argument
C-V View Next Screen
C-W Wipe Region
C-X C-X Command Prefix
C-Y Yank Killed Text
C-[see <ESC>
C-\ Delete Indentation

C-X <TAB> Set Tab Spacing
C-X C-B List Buffers
C-X C-C Exit to Command Level
C-X C-F Find File
C-X C-I (same as C-X <TAB>)
C-X C-M Delete Mode
C-X C-R Read File
C-X C-S Save File
C-X C-V View Next Screen Other Window
C-X C-W Write File
C-X C-X Exchange Point and Mark
C-X C-Z View Previous Screen Other Window
C-X . Set Indent Column
C-X = Where Am I
C-X 1 One Window
C-X 2 Two Windows
C-X B Select Buffer
C-X F Set Fill Column

C-X K Kill Buffer
C-X M Add Mode
C-X O Other Window
C-X ^ Grow Window

M- Delete Word Backward
M-<SPACE> see C-@
M-< Beginning of Buffer
M-> End of Buffer
M-A Backward Sentence
M-B Backward Word
M-C Capitalize Word
M-D Delete Word Forward
M-E Forward Sentence
M-F Forward Word
M-H Mark Whole Paragraph
M-K Kill Sentence Forward
M-L Lowercase Word
M-Q Fill Paragraph
M-R Replace String
M-S Center Line
M-T Transpose Words
M-U Uppercase Word
M-V View Previous Screen
M-W Copy Region
M-[Backward Paragraph
M-\ Delete Surrounding Whitespace
M-] Forward Paragraph

M-C-H see M-
M-C-K Kill Entire Line
M-C-R Query Replace String
M-C-W Append to Kill Buffer

Fill Mode Commands

space Auto Fill Space

Page Mode Commands

All printing characters: a-z, A-Z, 0-9, space, and
!"#\$%&'()*+,-./:;<=>@[]^_`{|}~`

Self-overwrite

C-A To First Non-White
C-B Backward Character on Line
C-E To Last Non-White
C-F Forward Character on Line
C-H Overwrite Character Backward
C-N Next Line Forced
C-P Previous Line Forced
C-Q Quote Next
C-X \ Delete Trailing Whitespace

USASCII Character Set

as modified for printing
and the inclusion of Meta characters

Decimal	Octal	Hex	Graphic	Name (Meaning) <English Text Reference>
0.	000	00	^@	NUL (used for padding) <NULL>
1.	001	01	^A	SOH (start of header)
2.	002	02	^B	STX (start of text)
3.	003	03	^C	ETX (end of text)
4.	004	04	^D	EOT (end of transmission)
5.	005	05	^E	ENQ (enquiry)
6.	006	06	^F	ACK (acknowledge)
7.	007	07	^G	BEL (bell or alarm) <BELL>
8.	010	08	^H	BS (backspace) <BS>
9.	011	09	^I	HT (horizontal tab) <TAB>
10.	012	0A	^J	LF (line feed) <LF>
11.	013	0B	^K	VT (vertical tab)
12.	014	0C	^L	FF (form feed, new page) <FF>
13.	015	0D	^M	CR (carriage return) <CR>
14.	016	0E	^N	SO (shift out)
15.	017	0F	^O	SI (shift in)
16.	020	10	^P	DLE (data link escape)
17.	021	11	^Q	DC1 (device control 1, XON)
18.	022	12	^R	DC2 (device control 2)
19.	023	13	^S	DC3 (device control 3, XOFF)
20.	024	14	^T	DC4 (device control 4)
21.	025	15	^U	NAK (negative acknowledge)
22.	026	16	^V	SYN (synchronous idle)
23.	027	17	^W	ETB (end transmission block)
24.	030	18	^X	CAN (cancel)
25.	031	19	^Y	EM (end of medium)
26.	032	1A	^Z	SUB (substitute)
27.	033	1B	^[ESC (escape, alter mode, SEL) <ESC>
28.	034	1C	^\ ^_	FS (file separator)
29.	035	1D	^]	GS (group separator)
30.	036	1E	^^	RS (record separator)
31.	037	1F	^_	US (unit separator)
32.	040	20		space or blank <SP>
33.	041	21	!	exclamation mark
34.	042	22	"	double quote
35.	043	23	#	number sign (hash mark)
36.	044	24	\$	dollar sign
37.	045	25	%	percent sign
38.	046	26	&	ampersand sign
39.	047	27	'	single quote (apostrophe)
40.	050	28	(left parenthesis
41.	051	29)	right parenthesis
42.	052	2A	*	asterisk (star)
43.	053	2B	+	plus sign
44.	054	2C	,	comma
45.	055	2D	-	minus sign (dash)
46.	056	2E	.	period (decimal point, dot)
47.	057	2F	/	(right) slash

48.	060	30	0	numeral zero
49.	061	31	1	numeral one
50.	062	32	2	numeral two
51.	063	33	3	numeral three
52.	064	34	4	numeral four
53.	065	35	5	numeral five
54.	066	36	6	numeral six
55.	067	37	7	numeral seven
56.	070	38	8	numeral eight
57.	071	39	9	numeral nine
58.	072	3A	:	colon
59.	073	3B	;	semi-colon
60.	074	3C	<	less-than sign
61.	075	3D	=	equal sign
62.	076	3E	>	greater-than sign
63.	077	3F	?	question mark
64.	100	40	@	atsign
65.	101	41	A	upper-case letter ALPHA
66.	102	42	B	upper-case letter BRAVO
67.	103	43	C	upper-case letter CHARLIE
68.	104	44	D	upper-case letter DELTA
69.	105	45	E	upper-case letter ECHO
70.	106	46	F	upper-case letter FOXTROT
71.	107	47	G	upper-case letter GOLF
72.	110	48	H	upper-case letter HOTEL
73.	111	49	I	upper-case letter INDIA
74.	112	4A	J	upper-case letter JERICH0
75.	113	4B	K	upper-case letter KAPPA
76.	114	4C	L	upper-case letter LIMA
77.	115	4D	M	upper-case letter MIKE
78.	116	4E	N	upper-case letter NOVEMBER
79.	117	4F	O	upper-case letter OSCAR
80.	120	50	P	upper-case letter PAPPA
81.	121	51	Q	upper-case letter QUEBEC
82.	122	52	R	upper-case letter ROMEO
83.	123	53	S	upper-case letter SIERRA
84.	124	54	T	upper-case letter TANGO
85.	125	55	U	upper-case letter UNICORN
86.	126	56	V	upper-case letter VICTOR
87.	127	57	W	upper-case letter WHISKEY
88.	130	58	X	upper-case letter XRAY
89.	131	59	Y	upper-case letter YANKEE
90.	132	5A	Z	upper-case letter ZEBRA
91.	133	5B	[left square bracket
92.	134	5C	\	left slash (backslash)
93.	135	5D]	right square bracket
94.	136	5E	^	uparrow (carat)
95.	137	5F	_	underscore

96.	140	60	`	(single) back quote (grave accent)
97.	141	61	a	lower-case letter alpha
98.	142	62	b	lower-case letter bravo
99.	143	63	c	lower-case letter charlie
100.	144	64	d	lower-case letter delta
101.	145	65	e	lower-case letter echo
102.	146	66	f	lower-case letter foxtrot
103.	147	67	g	lower-case letter golf
104.	150	68	h	lower-case letter hotel
105.	151	69	i	lower-case letter india
106.	152	6A	j	lower-case letter jericho
107.	153	6B	k	lower-case letter kappa
108.	154	6C	l	lower-case letter lima
109.	155	6D	m	lower-case letter mike
110.	156	6E	n	lower-case letter november
111.	157	6F	o	lower-case letter oscar

112.	160	70	p	lower-case letter pappa
113.	161	71	q	lower-case letter quebec
114.	162	72	r	lower-case letter romeo
115.	163	73	s	lower-case letter sierra
116.	164	74	t	lower-case letter tango
117.	165	75	u	lower-case letter unicorn
118.	166	76	v	lower-case letter victor
119.	167	77	w	lower-case letter whiskey
120.	170	78	x	lower-case letter xray
121.	171	79	y	lower-case letter yankee
122.	172	7A	z	lower-case letter zebra
123.	173	7B	{	left curly brace
124.	174	7C		vertical bar
125.	175	7D	}	right curly brace
126.	176	7E	~	tilde
127.	177	7F	^?	DEL (delete, rub out)

128.	200	80	~^@	Meta NUL
129.	201	81	~^A	Meta SOH
130.	202	82	~^B	Meta STX
131.	203	83	~^C	Meta ETX
132.	204	84	~^D	Meta EOT
133.	205	85	~^E	Meta ENQ
134.	206	86	~^F	Meta ACK
135.	207	87	~^G	Meta BEL

...

159.	237	9F	~^	Meta US
160.	240	A0	~ -	Meta space
161.	241	A1	~!	Meta exclamation mark

...

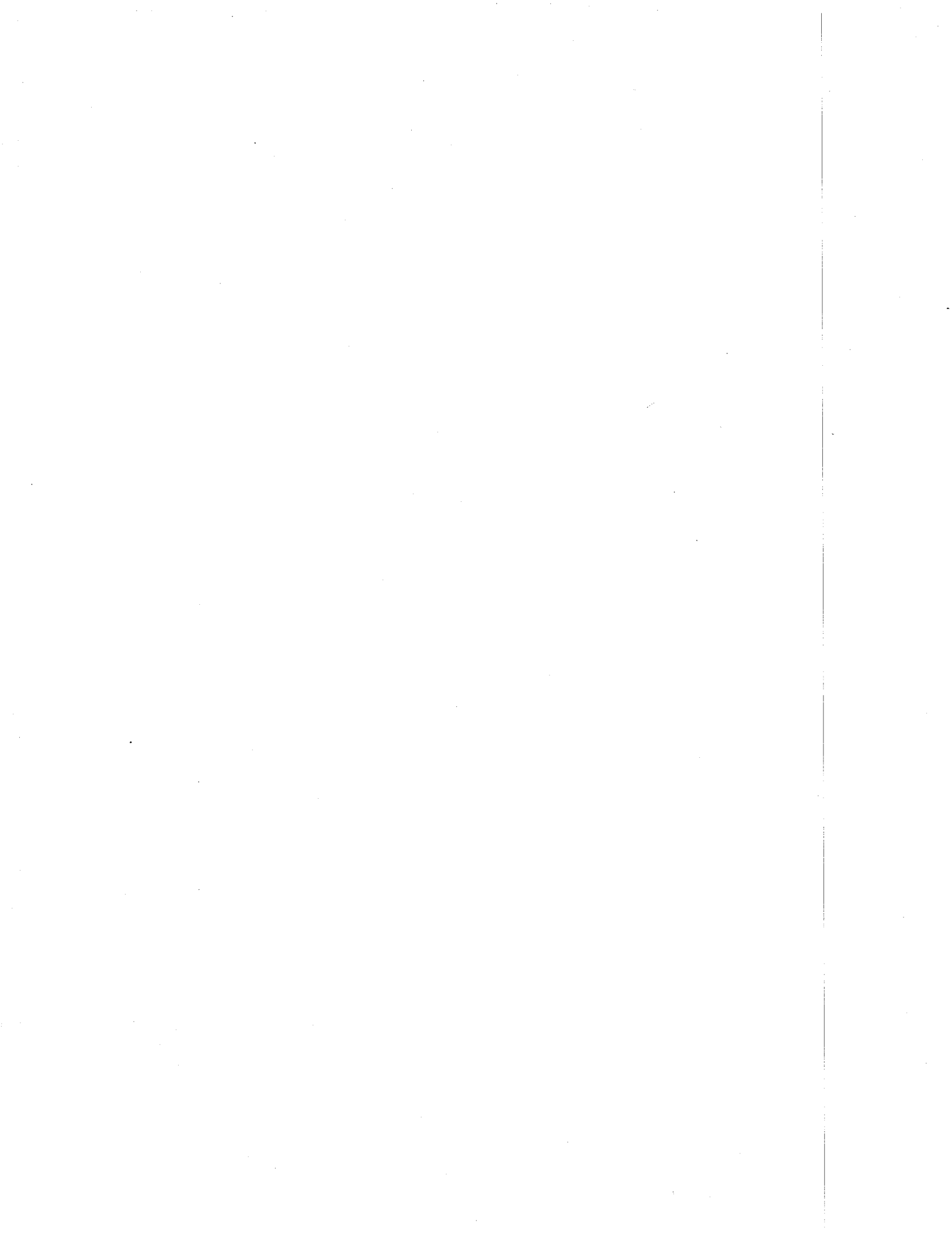
253.	375	FD	~}	Meta right curly brace
254.	376	FE	~~	Meta tilde
255.	377	FF	~^?	Meta DEL

Notes:

The "Meta" form of each character is created by adding 128 (decimal) to that character's ASCII value.

To prevent ambiguity, the following alternate forms can be used for printing:

94.	136	5E	^	can be printed as	^=
126.	176	7E	~	can be printed as	^~
222.	336	DE	^^	can be printed as	^^=
254.	376	FE	^^	can be printed as	^^~



Mince Internal Documentation
Table of Contents

Chapter 1 Program Logic Manual	1-1
1.1 Generalities	1-1
1.1.1 Notes on Data Abstractions	1-2
1.1.2 Quick Review of Mince	1-3
1.1.3 Code Structure	1-5
1.2 Specifics	1-6
1.2.1 Supplied Files	1-6
1.2.2 Coding and Documentation Conventions	1-7
1.2.3 Constants and Globals	1-8
1.2.4 Conditional Compilation Flags	1-13
1.3 Extending and Modifying Mince	1-14
1.3.1 An Example	1-14
1.3.2 On Changing Mince	1-15
1.3.3 Compiling and Linking Mince	1-16
1.3.4 Debugging Code	1-17
Chapter 2 Entry Points	2-1
2.1 Top Level and Redisplay Routines	2-1
2.2 User Level Buffer Description	2-4
2.3 Memory Allocation Abstraction	2-5
2.4 Queue Abstraction	2-5
2.5 Buffer Abstraction	2-6
2.5.1 Initialization and Buffer Manipulation	2-6
2.5.2 Inserting and Deleting Text	2-7
2.5.3 Beginning of Buffer, End of Buffer, and Basic Motion	2-8
2.5.4 Status and Complex Movement	2-9
2.5.5 Mark Manipulation	2-11
2.5.6 Reading and Writing Files	2-13
2.5.7 Private Routines	2-14
Chapter 3 Source Code	3-1
3.1 Control Commands: File COMML.C	3-1

3.2 Meta Commands: File COMM2.C	3-2
3.3 Control-X Commands: File COMM3.C	3-5
3.4 Support Routines: File SUPPORT.C	3-6
Chapter 4 The Terminal Abstraction	4-1
4.1 Initialization and Termination Routines	4-2
4.2 Cursor Positioning	4-3
4.3 Display Routines	4-4
4.4 Printing Text	4-5
4.5 Low Level Output and Keyboard Drivers	4-6
4.6 Internal Routines	4-8
Chapter 5 Theory and Practice of Text Editors	5-1
(NOTE: This chapter is internally sectioned into chapters, sections, and subsections.)	
1. Introduction	5-1
2. Memory Management	5-3
2.1 Data Structures	5-3
2.2 Marks	5-5
2.3 Interface Procedures	5-5
2.4 Buffer Gap	5-10
2.4.1 Gap Size	5-12
2.4.2 Multiple Gaps and Why They Don't Work	5-12
2.4.3 The Hidden Second Gap	5-12
2.5 Linked Line	5-13
2.5.1 Storage Comparison	5-14
2.5.2 Error Recovery Comparison	5-14
2.6 Multiple Buffers	5-15
2.7 Paged Virtual Memory	5-16
2.8 Editing Extremely Large Files	5-16
2.9 Scratchpad Memory	5-17
3. Incremental Redisplay	5-19
3.1 Line Wrap	5-20
3.2 Multiple Windows	5-20
3.3 Terminal Types	5-21
3.3.1 TTY and Glass TTY	5-21
3.3.2 Basic	5-21
3.3.3 Advanced	5-21
3.3.4 Memory Mapped	5-22
3.3.5 Terminal Independent Output	5-22

3.3.6 Echo Negotiation	5-23
3.4 Approaches to Redisplay Schemes	5-24
3.5 The Framer	5-24
3.6 Redisplay Algorithms	5-25
3.6.1 The Basic Algorithm	5-25
3.6.2 The Advanced Algorithm	5-27
3.6.3 Memory Mapped	5-29
3.7 Other Details	5-29
3.7.1 Tabs	5-29
3.7.2 Control Characters	5-29
3.7.3 End of the Buffer	5-30
3.7.4 Between Line Breakout	5-30
3.7.5 Proportional Spacing and Multiple Fonts	5-30
3.7.6 Multiple Windows	5-30
4. The Command Loop	5-31
4.1 Basic Loop: Read, Eval, Print	5-31
4.1.1 The Philosophy Behind the Basic Loop	5-31
4.2 Error Recovery	5-32
4.3 Arguments	5-33
4.3.1 Prefix Arguments	5-33
4.3.2 String Arguments	5-34
4.3.3 Positional Arguments	5-35
4.4 Rebinding	5-35
4.4.1 Rebinding Keys	5-36
4.4.2 Rebinding Functions	5-36
4.5 Modes	5-37
4.5.1 Implementing Modes	5-38
4.6 Kill and UnDo	5-38
4.7 Implementation Languages	5-39
4.7.1 TECO	5-39
4.7.2 Sine	5-40
4.7.3 Lisp	5-40
4.7.4 PL/1, C, etc.	5-40
4.7.5 Fortran, Pascal, etc.	5-40
5. User Interface Hardware	5-41

5.1 Keyboards	5-41
5.1.1 Special Function Keys and Other Auxiliary Keys	5-42
5.1.2 Extra Shift Keys	5-42
5.2 Graphical Input	5-42
5.2.1 How It Can Be Used	5-43
5.2.2 Devices: TSD, Mouse, Tablet, Joystick	5-43
6. The World Outside of Text Editing	5-45
I. Annotated Bibliography	5-47
6.1 Emacs Type Editors	5-47
6.1.1 ITS EMACS	5-47
6.1.2 Lisp Machine Zwei	5-48
6.1.3 Multics Emacs	5-48
6.1.4 MagicSix TVMacS	5-49
6.1.5 Other Emacs	5-49
6.2 Non-Emacs Display Editors	5-50
6.3 Structure Editors	5-51
6.4 Other Editors	5-52
II. Some Implementations of Emacs Type Editors	5-55
III. Partial Emacs Command List	5-57

Chapter 1

Program Logic Manual

1.1 Generalities

This is the program logic manual for Mince. It discusses a variety of topics: First, it reviews the basic terminology used to discuss the editor and identifies the parts of Mince. Second, it explains the conventions and structure in the implementation. Third, it discusses the file MINCE.GBL and reviews the use for each of the variables found there. Finally, it warns about some potential pitfalls in modifying the existing code. These topics are not discussed in any particular order. Throughout the manual comments on customizing Mince will be made. These comments will be indented, as in:

First customization note. When you are writing a new command, it is a good idea to make a copy of an existing function which does a related task and modify that copy. Writing new commands from scratch is much harder.

This chapter cannot be read by itself. In order to get a full understanding of the structure of Mince and how to modify it, the rest of the manual is needed. First and foremost is the Mince User's Guide. It can answer questions about general concepts: for example, why you want to use GetArg as opposed to Ask. It explains what Mince looks like to the user.

Next is the Complete Command List. That document gives the definition of each command in words. You can then look at the C source code and see how that command is actually implemented. That chapter is also useful in reverse. If you are looking at some code and can't figure it out, looking at the English description can be of great help. Both the Mince User's Guide and the Complete Command List appear in the Mince User's Manual.

The third chapter of use is "Theory and Practice of Text

Editing." This chapter describes the structure of a text editor and spends a lot of time discussing text buffers and the redisplay, two major parts of Mince for which source code is not supplied. In spite of the fact that the chapter was written only last May (1980) and Mince was written in October (1980), the underlying software technology has changed substantially. However, the chapter does provide a solid base upon which to understand Mince and the general approaches presented are still valid. Note that the buffer interface described in that chapter are not the same as those actually used in Mince.

The fourth item of interest is the Entry Point Documentation. This chapter documents each entry point in Mince for which source code is not supplied. It gives enough information for each entry point to allow you to use that entry point.

The fifth item of interest is the Source Documentation. It explains in great detail some of the more complicated routines for which source code is supplied. This item also describes the interface for each routine in SUPPORT.C. Understanding what these routines do is often the key to understanding a routine that they are used in. An extreme example is DoReplace. The entire Query Replace command (MQryRplc in COMM2.C) merely calls DoReplace(TRUE).

The final useful item for understanding Mince is the source code itself. In it, you see a concrete implementation of what all this documentation can only talk about in abstract terms.

1.1.1 Notes on Data Abstractions

Data abstractions are programming tools. Like any tool, they can either be appropriate to a given situation or not and they can be both correctly and incorrectly applied. We used them in Mince because they were appropriate and aided us in the development.

A data abstraction is a collection of subroutines and data. Only those subroutines are allowed to access that data, and that data serves as the sum knowledge that the subroutines have about the "world." An imaginary line can be drawn around these subroutines and data. All knowledge about the internal representation of the data is contained within this line.

The data abstraction is manipulated by calling the subroutines and passing them arguments. A subroutine is defined for every operation that can be done with the data.

The data abstraction is defined by its interface to the outside. Any internal arrangement of data and procedures that implements the interface is acceptable and DIFFERENT IMPLEMENTATIONS ARE COMPLETELY INTERCHANGABLE! This last property is of critical importance. It allows substantial revisions of programs while confining these changes to only those procedures that are directly affected. For example, while developing Mince, we rewrote the buffer abstraction so as to provide virtual memory. No command routines had to be modified at all. Without the high degree of information isolation provided by the buffer abstraction, this change would have proven much more difficult.

In summary, a data abstraction is a way to contain information. It does this by hiding all of the internal representations and only showing the "outside world" a clean, simple interface.

1.1.2 Quick Review of Mince

Mince is a multi-buffer in-memory text editor. This means that it edits a copy of a file by reading it into a text buffer. This buffer is usually in main memory (i.e. RAM). Having more than one buffer allows you to edit several files at once by copying each of them into a different buffer, then switching back and forth from buffer to buffer. Since the available main memory space (RAM) may not be large enough to store all of the text which can be read into buffers, the existence of additional main memory is simulated by a virtual memory system, which copies parts of buffers from main memory to disk (i.e., secondary storage) and vice versa as necessary. (The buffer abstraction, whose entry points are described elsewhere, handles both of these functions.)

Each buffer has a Point associated with it. All editing changes take place at the Point. There are also a number of marks which can be placed anywhere in any buffer. A mark will stay in the same place with respect to the surrounding text no matter what changes go on around it. Each buffer always has a distinguished mark associated with it. It is this mark that is referred to as "the Mark" in the user documentation.

A buffer also has a mode list. Modes are ways of tailoring the command set on a per-buffer basis. Modes are coded in C and represented in Mince as the differences between the default command set and the desired one.

Each keystroke that is typed invokes a function to implement the meaning associated with (or "bound to") that key.

Customization Note. When devising modes, it helps a lot if they don't rebind an existing command or, if they do, that the same command key is not also rebound by another mode. Page mode and Fill mode have this problem: if you invoke Fill mode first, after Page mode is added it destroys the definition of Auto Fill Space which was bound to the Space key. Such things are annoying at best.

Mince is implemented as an editor within an editor. The outer editor interfaces to the user and reformats his or her desires so as to be executable by the inner editor, known as the buffer abstraction. It might help to think of its interface as a user's manual for the inner editor.

The buffer abstraction sub-editor takes all memory left over after the code, globals, and operating system have taken what they need and divides it into 1K pages. These pages are used to store the contents of the buffer. They are swapped between memory and the swap file on an LRU (least recently used) basis.

In any virtual memory scheme, a page must be swapped out of main memory to make room for the desired page. An LRU scheme is one where the page that was least recently accessed is the one that is chosen to be swapped out. Swapping out a modified page (the copy in memory is different from the copy on disk) requires that the page be written to disk. Swapping out an unmodified page doesn't require any activity. Thus, the LRU scheme that Mince uses has been modified to try to swap out unmodified pages first as doing so takes less time. That is also why Mince swaps out modified pages (making them unmodified) when it is otherwise idle; when you start editing again, it has less work to do.

BFlush is used to implement the delayed write-through. If the system is idle, pages are written through one by one to the swap file until everything out there is current. Note that there are two types of modified pages. One type is what the user sees: a page (buffer) has had an insert or delete performed upon it. The other type does not imply insertion or deletion. Rather, it means that the page is different from the copy of it on the swap file. It therefore must be written to the swap file before its memory can be reused. The latter case happens, for example, when reading a file in for the first time. Although the buffer is unmodified, the pages have to be swapped through to disk.

Redisplay is the process of updating the user's terminal's screen to reflect the current contents of the buffer. Mince

command routines need know nothing about this process as it is handled automatically by the sub-editor. The redisplay process is invoked each edit command cycle, just before the editor asks for a command. If a command has been entered, or if one is entered as redisplay occurs, the redisplay is aborted and the command is executed immediately.

The redisplay operates by scanning the buffer and comparing its current contents with the redisplay's internal model of what is on the screen. The internal model consists of an array of special screen marks, one for each screen line. Each screen mark has a modified flag associated with it. Whenever an insert or delete operation takes place, the buffer abstraction automatically sets the modified flag on the associated screen mark. The redisplay process thus can determine which parts of the screen could have been affected.

1.1.3 Code Structure

The basic edit loop is as is discussed in "Theory and Practice of Text Editors," chapter Five. Function Main calls Setup and then calls Edit. Setup does a lot of initialization, but the important thing is that it calls SetModes.

SetModes is a critically important function. It is called at initialization, when switching buffers, and when adding or deleting modes. It is responsible for initializing the command bindings to their new values. It fulfills this task by calling, in turn, finit1, finit2, and finit3 which set up the default bindings for Control, Meta, and Control-X commands, respectively. It then modifies these defaults by going down the mode list for that buffer and performing the tailoring specific to each mode.

Customization Note. When adding or removing a mode from the code, the change must occur in two places. First is in SetModes. The other place that the change must take place is in CheckModes in SUPPORT.C.

After SetModes has been called, Setup returns and then Edit is called. Edit performs an IncrDsp (incremental redisplay), waits for a character, and dispatches to the command routine that the character is bound to. (A "command routine" is any procedure which can be called directly from a dispatch table.) Arg is set to 1 and Argp is set to FALSE.

Any further calls to SetModes will be from commands (e.g., Switch Buffers).

At this point, each command can assume the following environment:

Arg is set to 1
Argp is set to FALSE
Lfunc points to the function bound to the command that was executed before this one (the "last function" executed)
Cmnd is set to the character that was typed by the user to generate this command

Note that "this point" does not always exist. If a C-U (Universal Argument, MArg in COMML.C) is typed, it will eventually dispatch again to the commands. In that case, Arg will in general not be 1, Argp will be TRUE, and Cmnd will still be the character that was bound to you. Similar changes happen with the Meta and Control-X dispatch functions. Note that 128 <= Cmnd <= 255 indicates a Meta command and 256 <= Cmnd <= 383 indicates a Control-X command.

If Arg is greater than one when you return back to edit, edit will decrement it and call you again.

1.2 Specifics

1.2.1 Supplied Files

There are several source files supplied with Mince. They are:

BINDINGS.C	Source code for the key binding functions finit1, finit2, finit3, and SetModes.
COMML.C	Source for Control commands. COMML, COMM2, and COMM3 have the routines listed in the same order as appears in the full command list, i.e. in the same order as the ASCII collating sequence for the keys to which they are bound.
COMM2.C	Source for Meta commands.
COMM3.C	Source for Control-X commands.

SUPPORT.C Source for support routines used by the commands. They are listed in alphabetical order.

1.2.2 Coding and Documentation Conventions

The Entry Point Documentation, the Source Code Documentation, and the Terminal Abstraction Documentation follow certain conventions. First, the name and type of each argument is given; the return argument is only given if the routine specifically returns a meaningful value. (In C, everything returns a value.) Second, each global variable that the routine accesses is listed in those routines for which the source code is not given. Here is a guide to interpreting the ways that globals are used:

Exports means that that routine sets the global for other routines' use.

Imports means that that routine reads the value of that global.

Private means that no other routines should look at or set that variable.

Updates means that that routine both reads and writes the value of that global.

Uses means that that routine bashes any existing value and leave a garbage value in that global.

Note that there is no listing of which globals any of the command routines access. In general, they utilize globals heavily and a quick check of the source code can tell you which of them a particular routine uses.

The upper/lower/mixed caseness of names also has meaning.

UPPERCASE names are constants. They are initialized with #defines and are only initialized in MINCE.GBL.

MixedCase names are procedures. A capital letter usually indicates the start of a new word (in lieu of a space or underscore).

lowercase names are variables or procedures. When procedures, they usually name a procedure used only locally.

Within written English text, a lowercase name will often be Capitalized in order to facilitate its recognition as a variable or procedure name.

The first letter of a procedure name usually has meaning as well. Note that the routines in SUPPORT.C ignore this convention completely.

letter	indicates that the procedure is part of the...
A	dynamic memory Allocator
B	Buffer abstraction
C	top level buffer abstraction that makes the buffer abstraction Compatible with the Mince user-visible view of buffers
M	Mince command set
Q	FIFO Queue maintainer
T	Terminal abstraction

1.2.3 Constants and Globals

This section discusses the contents of the file MINCE.GBL. It briefly covers the meaning of each of the constants and globals.

Basic constants and variables:

TRUE	(-1)	
FALSE	0	
NULL	0	-- the null pointer
HOME	0,0	-- TSetPoint(HOME) puts you at the upper left corner
FORWARD	(-1)	-- alternates names for TRUE and FALSE to enhance readability in some places
BACKWARD	0	
SWAPFNAM	"mince.swp"	-- two places where it looks
SWAPLFNAM	"a:mince.swp"	for the swap file
INPUT	0	-- mode for file input
OUTPUT	1	-- and output
UPDATE	2	-- and update
FILMAX	15	-- maximum length of a filename
STRMAX	40	-- maximum length of a search string
MODEMAX	20	-- maximum length of the mode name string
MAXMODES	4	-- maximum number of modes (per buffer)
BUFNAMMAX	9	-- maximum length of a buffername
BUFFSMAX	7	-- maximum number of buffers

```

mark      -- user settable mark in the current buffer

arg       -- the numeric argument to a command
argp     -- TRUE if an explicit argument was entered,
          FALSE otherwise

lcol     -- column that Next Line and Previous Line
          try to leave you in

psstart  -- a mark placed one character before the
          start of the screen
sstart   -- a mark placed at the start of the screen
send     -- a mark that tries to be placed at the end
          of the screen (not valid if
          redisplay was interrupted)

cnt, tmp  -- scratch variables for local use by
          commands or support routines. Watch
          out for calling and called routines
          that use the same variable!

(*functs[3*128]) () -- the key bindings table
(*lfunct) ()       -- the previous command invoked

          for commands that deal with margins...
fillwidth -- the first column text can't be in
indentcol -- the first column text can be in

strarg[STRMAX]   -- previous search string
mode[MODEMAX]   -- current mode name sting
namearg[BUFNAMMAX] -- previous buffer name

pnt_row  -- screen line the Point was on in the
          last redisplay
stat_col -- column that the statistics (-%-, etc)
          begin in

abort    -- set it to TRUE if you want to exit
          the editor (abort out of the
          command loop)
cmnd     -- current command character (128 <=
          cmnd <=255 is a Meta command, 256
          <= cmnd <= 383 is a Control-X
          command)

cbuff    -- index of the current buffer in the
          "buffs" structure
del_buff -- buffer descriptor of the delete

```

```

                                buffer for use with the buffer
                                abstraction (e.g., BSwitchTo)
tmark                          -- scratch variable used to hold a mark

                                user visible buffer structure
struct cbuffer {
    bbuff                        -- buffer descriptor for use
                                with the buffer abstraction
    bmark                        -- the user settable mark
    bname[BUFNAMMAX]            -- the buffer name
    fname[FILMAX]               -- the associated filename
    bmodes[MAXMODES]            -- the list of mode ids for
                                this buffer
} buffs[BUFFSMAX]

```

Terminal Abstraction constants and variables:

```

ROWMAX      60  -- maximum configurable # of rows
COLMAX      132 -- maximum configurable # of columns

NUL  '\0'  -- ASCII NULL character (0 decimal, ^)
BELL '\7'  -- ASCII BELL character (7 decimal, ^G)
BS   '\10' -- ASCII BACK SPACE character
        (8 decimal, ^H)
TAB  '\11' -- ASCII HORIZONTAL TAB character
        (9 decimal, ^I)
LF   '\12' -- ASCII LINE FEED character
        (10 decimal, ^J)
CR   '\15' -- ASCII CARRIAGE RETURN character
        (13 decimal, ^M)
ESC  '\33' -- ASCII ESCAPE character
        (27 decimal, ^[])
DEL  '\177' -- ASCII DELETE character
        (127 decimal, ^?)
NL   '\212' -- character that Mince uses to mean
        Newline (138 decimal, ~^J)

KBBUFMAX  80  -- maximum number of typeahead
        characters

prow      -- position of the screen point
pcol
srcw      -- position of the cursor
scol

tabincr   -- number of columns between tab stops

```

```

tthrow          -- row number of the saved screen row

clrcol[ROWMAX] -- for each row, the number of the
                  column after the last non-blank
                  character

tline[COLMAX]   -- the saved screen line (for redisplay)

lindex          -- temporary pointer into tline

struct {        -- keyboard input queue structure
    head
    tail
    bottom
    top
    space[KBBUFMAX]
} kbdq

```

Terminal Abstraction variables that are read from the swap file header. The swap file header was written to disk by Config.

```

NOPAD          0    -- for "padp", if Padp==NOPAD, no
CHARPAD        1    padding is necessary, if Padp==
DELAYPAD       2    CHARPAD, pad with characters, if
                    Padp==DELAYPAD, pad with wait loop

FIRST          255  -- possible arguments to put_coord.
SECOND         0    Tells whether to put out the first
                    or the second coordinate this time.

                structure describing the terminal
struct termdesc {
    ctrlz      -- ^Z to tell the world to stop reading
                the file, as it will contain what
                looks like garbage
    nrows      -- number of rows on the terminal
    ncols      -- number of columns on the terminal
    rowbias    -- bias to add to the desired row and
    colbias    -- column when doing cursor positioning
    rowfirstp  -- TRUE if the row should be sent first
    compp      -- TRUE if the row and column should be
                bitwise complemented after biasing
                and before sending
    binaryp    -- TRUE if the row and column should be
                sent in binary, FALSE if they
                should be sent in ASCII
    padp       -- tells how to pad commands
    padchar    -- what character to pad with for

```

```

                                padp==CHARPAD
nhclpad  -- amount of padding to do for home and
          clear screen
ncleolpad -- amount of padding to do for clear to
          end of line
ncpospad -- amount of padding to do for cursor
          positioning
ncleowpad -- amount of padding to do for clear to
          end of screen

```

For each of the operations that follow, this is the index and length of the character string to send:

```

struct str {
    idx  -- index
    len  -- length
} hcl    -- home and clear screen
    cleol -- clear to end of linen
    cleow -- clear to end of window (screen)
    cpos1 -- prefix string for cursor
          positioning
    cpos2 -- string to separate the two
          coordinates
    cpos3 -- postfix string for cursor
          positioning
    bell  -- ring the terinal bell
    init  -- intialize the terminal
    deinit -- leave the terminal in a
          reasonable state

strspc[73] -- the space that the above
            operations index into. It
            must be 73, as it fills
            out the disk block.

} terminal

```

I/O port descriptions:

```

struct portdesc {
    biosp    -- TRUE if the bios is used.  If FALSE,
              the rest is relevant:
    dataport -- number of the data port
    statport -- number of the status port
    datamask -- ANDED with incoming data
    readymask -- ANDED with status to deterine whether
               a character is waiting or the port
               is ready for output
    polarity -- polarity of the relevant bit.  TRUE

```

```

    } inport      if "1" bit means the port is ready
                  --for both input and output ports
    output

```

Random parameters:

```

prefrow      -- preferred cursor row
fillinit    -- initial fill column
tabinit     -- initial indent column
indentinit  -- initial tab increment
mhz         -- processor speed in tenths of
              megahertz
delaycnt    -- delay constant for echoing "Meta: ",
              etc. as well as wait time before
              swapping starts
npages      -- number of pages in swap file. Must
              be a multiple of 8
swapbase    -- base of the actual swap area in
              sectors

```

Spare area for patches:

```

spare[10]    -- ten integers' worth

```

Note that you cannot alter these declarations in any way. However, the Spare variables are available for use by any routines that you write.

1.2.4 Conditional Compilation Flags

Mince has conditional compilation flags scattered throughout the source code. These flags are used to tailor the Mince source code for a variety of machines and operating systems. The flags are:

```

UNIX        -- indicates the operating system. only
RSX         one of these can actually be on
CPM

SUSER      -- single user system
LARGE     -- extra command memory available
TYPEAHEAD -- typeahead is detectable

```

Note that for CP/M systems, the only flag that can be altered is the LARGE flag.

1.3 Extending and Modifying Mince

1.3.1 An Example

Let us uncover some of the potential pitfalls and see how all of this hangs together by writing an example function. This function is a sort of poor man's detabify. It's called MDeTab and it will find the next Tab character and replace it with the number of spaces that it represents.

A noticeable amount of implicit knowledge was used in the above paragraph. First, the knowledge of how routines are named indicated that the function name should be prefixed with "M". ("M" stands for "Mince.") Second, the knowledge that the general function (detabification) is useful is implicit in selecting this particular example. Third, the knowledge of how this function could fit in with the "Mince philosophy" to serve as the foundation for a "Detabify Region" command helped to shape the definition. The "Mince philosophy" is something that is gradually acquired by writing a (possibly large) number of commands and trying to fit them in with the existing structure.

The first step is to establish the algorithm. This step is not as forbidding as it sounds. All that it implies is that we rewrite the above description in a more formal way:

```
find the next Tab
figure out how wide it is
replace the Tab with that many spaces
```

Or, yet more formally:

```
search forward through the buffer for a Tab
figure out how wide it is
delete the Tab
insert the correct number of spaces
```

In C/Buffer Abstraction "Language" this would be:

```
BCSearch(Tab)
width=TWidth(column, Tab)
BDelete(1)
call BInsert(" ") WIDTH times
```


Note that this is NOT a finished function and, as it stands, it will not work. It is left at this stage to point out different ways of making two coding decisions. First, the call to TWidth might have been coded as a BGetCol, a move backwards one character, and a second call to BGetCol. The difference between the returned values in the column positions is the answer (and is the same number that TWidth will return). Second, the insertion of the WIDTH spaces can be by an explicit for- or while- loop or it can be via a call to Indent. This example illustrated that there is probably a function in Mince that directly does what you want; your job is to ferret it out.

The finished function looks like this:

```
MDeTab()          /* change a Tab to blanks */
{
    if (!BCSearch(TAB)) return;
    BMove(-1);
    SIndent(TWidth(BGetCol(),TAB));
    BDelete(1);
}
```

The if statement puts in a very important check: if there is no Tab, we don't do anything. Checks of this sort are very important in finished code. However, they are not relevant to the definition and so were left out of the earlier discussion. We then go backwards over the Tab. Thus, the BGetCol will return the desired column (the one you are in just before the Tab is "printed"). TWidth takes this column and returns the width of the Tab. SIndent puts in that many spaces. The BDelete then deletes the Tab.

If we were to write a Detabify Region command, this function would serve as a good base. It does need some touching up, however. First, it should be passed a mark which was placed at the end of the region. MDeTab would then not do anything if the BCSearch left you after the mark. Second, MDeTab would probably return either TRUE or FALSE, depending upon whether it did anything. The Detabify Region command could check this flag to determine whether to continue on in the loop.

1.3.2 On Changing Mince

As the previous example indicated, there is a lot to know before you change or extend Mince. The best way to acquire some of this knowledge is to first become an expert in using Mince. After all, it is wasteful to write a Center Line command if one

already exists and you merely didn't know about it. That knowledge will help you to figure out how existing code works and it will also help you have your modifications fit in with the "Mince philosophy." Unless you are reworking the entire command set and user interface, users will be much happier if any changes are in line with the existing philosophy of the program. It is easier to learn and be happy with an undesirable philosophy that consistently implemented than with the same philosophy that has been changed here and there so as to be "less undesirable." Of course, we feel that our philosophy is not undesirable...

1.3.3 Compiling and Linking Mince

There are a number of points to consider when compiling and linking a version of Mince. First, the BDS C compiler (Version 1.42 or higher) must be used for compiling Mince.

The object code files (.CRL) are distributed in two forms. Both forms were compiled with the `-e` option. (We estimate a 25% increase in size without this option.) The object code files for which source code is not supplied are:

MINCE.CRL	LMINCE.CRL
UTIL.CRL	LUTIL.CRL
VBUFF1.CRL	LVBUFF1.CRL
VBUFF2.CRL	LVBUFF2.CRL

The normal-named versions use `-e7900` and the versions starting with "L" use `-e8100`. The extra 2K of space allows room for adding functions to Mince. Note that there is essentially no extra space in the `-e7900` versions so if you add code there, you must take out something else. If more space is required, contact us and we will generate a special version.

Note: when compiling `SUPPORT.C`, the `-r` option must be used to prevent symbol table overflow (use `-r10`).

When linking, use the linker supplied by Mark of the Unicorn (called "L2") and NOT the linker supplied by BDS. (Among other reasons, our linker saves 10% of code space.) This linker is experimental and is not guaranteed to link any arbitrary C program. It will, however, link any software supplied by us.

The following command line will successfully link a Mince (it can be found in `ML.CMD`):

```
l2 mince bindings comm1 comm2 comm3 vbuff1 vbuff2 -l
```

support term util

Almost nothing else will. About all that you can do is to split or include new command files, in which case be sure that they are before the "-l". If you would like to know more about the linker or have any problems, contact us.

1.3.4 Debugging Code

So, you have written some new functions and would like to debug them. Debugging a display editor is not quite the same as debugging anything else. For one thing, the screen has a tendency to rearrange itself on you...

The best method to use is a modification of the basic tracing techniques that every programmer has used: put in a print statement. The function Debug is the print statement and you simply have to call it every place that you would like to print a value or message. Debug does the following:

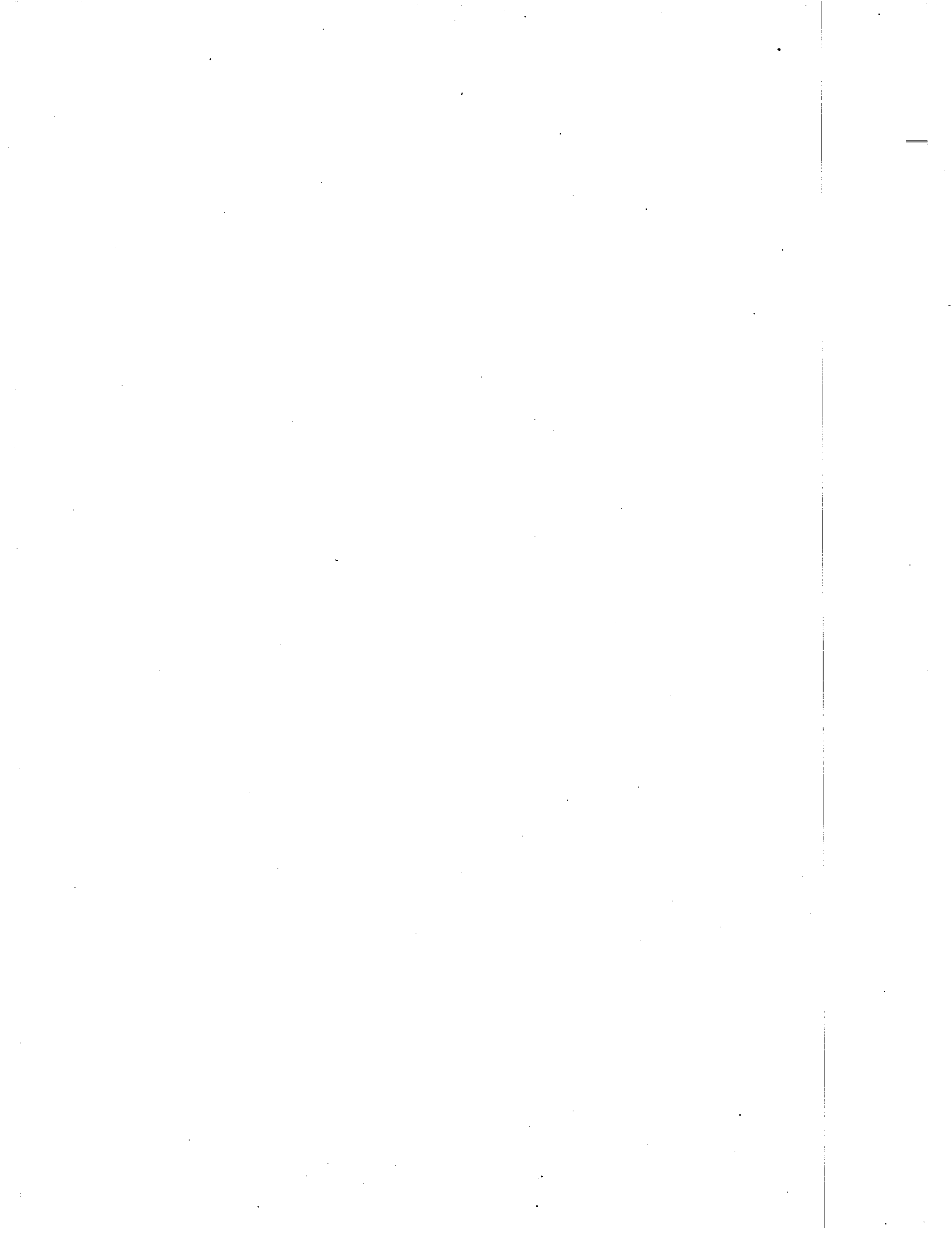
```
it prints a message
it prints the value of an integer
it does an redisplay (so you can see what the buffer
  looks like)
it waits for you to type a character, and returns
  TRUE if the entered character was a Delete or
  BackSpace, FALSE otherwise
```

It is a good idea to sprinkle a lot of calls to Debug in any suspicious code, especially in infinite loops. Each message should be different (to allow you to see what part of the loop you are actually in. In this manner, you can see the "intermediate results" of your function. You can also get out of the function to try something else by doing:

```
if (Debug("I am here",cnt)) {
    arg=0;
    return;
}
```

Don't even think of trying to use a conventional debugger with Mince. You haven't got a chance.

Good Luck!



Chapter 2

Entry Points

2.1 Top Level and Redisplay Routines

```
main(argc,argv)
  int argc
  char *argv[]
```

This is the command line entry point.

```
setup(iargc,iargv)
  int iargc
  char *iargv[]
```

Exports the globals fillwidth, del_buff, indentcol, namearg, stringarg, and tabincr.

Initialize the editor. It is called immediately by main and may not be reinvoked. Fillwidth is initialized to the default fill width and indicates the first column that characters may not appear in while filling text. Indentcol is initialized to the default indent column and indicates the first column that characters may appear in while filling text. Tabincr is initialized to the default tab increment and indicates the number of columns from one tab stop to the next. Namearg is initialized to the null string and retains the default used for switching buffers. Stringarg is initialized to the null string and retains the last search string. Del_buff is a buffer descriptor and

defines the kill buffer.

The last act performed is to call the routine UInit to perform any initialization desired by the user.

edit()

Exports the globals abort, arg, argp, cmnd, and lfunct.
Imports the global functs.

The basic editor loop. It reads a command, dispatches it, and invokes redisplay. It may be called recursively and setting the global abort to TRUE will exit the current invocation. Argp indicated the presence (TRUE) or absence (FALSE) of an argument to a command and is initialized to FALSE. Arg is the actual argument value and is initialized to 1. Cmnd contains the key that was typed combined with any prefix codes. Lfunct is the address of the procedure invoked by the previous command.

NewDsp()

Clears the screen, does a ScrnRange, and forces redisplay.

IncrDsp()

Imports the globals psstart, send, sendp, and sstart.

Performs a redisplay. Updates the screen, one line at a time, to reflect the current state of the buffer. If a character is typed during redisplay, it aborts after finishing a line. Upon successful completion, Send is placed at the actual end of the window. It also calls ModeFlags.

int

InnerDsp(from, to, pmark)
int from, to, pmark

Exports the globals send, sendp, and tline.
Imports the globals cur_cptr and terminal.
Private globals lindex, pnt_row and tline.
Updates the global pcol.

Redisplays a single window for IncrDsp. The window runs from screen lines From to To. If Pmark is non Null, returns the row that the mark is on. This routine should not be called outside IncrDsp.

ScrnRange()

Exports the globals psstart, sstart, and send.

Centers the window so that the Point is on PREFLINE. Sstart is a mark which is placed at the start of the window. Send is a mark which approximates the end of the window. Psstart is a mark which is placed one character before the start of the window.

int WHeight()

Imports the globals divide and topp.

Returns the height of the current window in lines.

int PrefLine()

Imports the global prefrow.

Returns the line within the current window that the user prefers the Point to be on. This value is linearly dependent upon the position within the window of the value of the "preferred row" parameter given in Config.

ModeLine()

Exports the global stat_col.
Imports the global buff_s, cbuff, and mode.

Displays the mode line. Stat_col indicates where to start displaying the mode flags.

2.2 User Level Buffer Description

The data structure is:

```
struct cbuffer {
    int bbuff, bmark;
    char bname[BUFNAMMAX], fname[FILMAX], bmodes[MAXMODES];
    } buffs[BUFFSMAX];
```

```
int
CMakeBuff(buffername, filename)
    char *buffername, *filename
```

Creates a buffer named Buffername with an associated filename Filename. It also sets the modes to no modes, creates a mark, and sets bbuff to be a buffer descriptor of a new buffer. Returns the index of the newly created buffer, or -1 if it failed. Buffername must be unique. If it is not, the results are undefined.

```
CSwitchTo(bufferindex)
    int bufferindex
```

Exports the globals cbuff and mark.

Makes the buffer selected by Bufferindex the current buffer, calls SetModes, and makes the mark associated with that buffer the global mark. Cbuff is a bufferindex and is the index of the current buffer.

```
int
CFindBuff(buffername)
    char *buffername
```


Returns the index of the buffer whose name is Buffername or -1 if there is no such buffer.

2.3 Memory Allocation Abstraction

The following routines implement a dynamic storage allocator and are used internally by the buffer abstraction. They cannot be used outside of the buffer abstraction.

AAlloc	ACoalesce	AFindNext
AFree	AInit	ALen
APrtWrld	ASpace	

They require the following globals:

```
int *Abegin, Asize, *Aend, *Acap
```

2.4 Queue Abstraction

The following routines implement a FIFO character queue. They all use the following definition of a queue.

```
struct queue {
    char *qhead, *qtail, *qtop, *qbottom, qspace[size]
}
```

```
QInit(queue_pointer, size)
struct queue *queue_pointer
int size
```

Format a queue structure that you have allocated and pass to QInit. Size tells the initializer how big the queue should be. You must allocate the space yourself.

This space includes both the space for the queue and the space for the queue header. Size here and the size in the structure declaration are the same.

```
char
QGrab(queue_pointer)
    struct queue *queue_pointer
```

Returns the next character on the queue. Results are undefined if the queue is empty.

```
QShove(char, queue_pointer)
    char char
    struct queue *queue_pointer
```

Places Char onto the queue. Results are undefined if the queue is full.

```
FLAG
QEmpty(queue_pointer)
    struct queue *queue_pointer
```

Returns TRUE if the queue is empty; FALSE otherwise.

```
FLAG
QFull(queue_pointer)
    struct queue *queue_pointer
```

Returns TRUE if the queue is full; FALSE otherwise.

2.5 Buffer Abstraction

2.5.1 Initialization and Buffer Manipulation

BInit(swap_file_descriptor)
int swap_file_descriptor

Initializes the buffer abstraction and tells it to use the indicated file as the swap file.

int
BCreate()

Creates a new buffer and returns its buffer descriptor. Returns NULL if no more buffers are available or there is no more memory.

BSwitchTo(buffer_descriptor)
int buffer_descriptor

Makes the buffer indicated by Buffer_descriptor the current buffer.

BDelBuff(buffer_descriptor)
int buffer_descriptor

Deletes the buffer indicated by Buffer_descriptor. You cannot delete the current buffer (indicated by the global Cbuff), and you will get an error message if you try.

2.5.2 Inserting and Deleting Text

BInsert(character)
char character

Inserts Character at the Point. It gives an error message and does nothing if there is no more available (virtual) memory.

BDelete(quantity)
unsigned quantity

Deletes Quantity characters forward from the Point.

BDelToMrk(mark)
int mark

Deletes all text between the Point and the passed mark. The passed mark must be in the current buffer, and you will get an error message if it is not.

BCopyRgn(mark,buffer_descriptor)
int mark
int buffer_descriptor

Copies the text between the Point and the passed mark into the indicated buffer, inserting the text at that buffer's Point. The passed mark must be in the current buffer, and you will get an error message if it is not. You cannot copy into the same buffer that you are copying from, and will get an error message if you try. Leaves the destination buffer's Point after the inserted text. If it runs out of available memory, it will abort after having copied as much as it can and print the message "Swap file full".

2.5.3 Beginning of Buffer, End of Buffer, and Basic Motion

BToStart()

Moves the Point to the beginning of the buffer.

BToEnd()

Moves the Point to the end of the buffer.

BShoveIt()

Places the buffer in a repeatable, invalid state. It is used internally by redisplay. Results are undefined if it is invoked outside of redisplay.

FLAG
BIsStart()

Returns TRUE if the Point is at the beginning of the buffer, FALSE otherwise.

FLAG
BIsEnd()

Returns TRUE if the Point is at the end of the buffer, FALSE otherwise.

BMove(dist)
int dist

Moves the Point Dist characters. Dist may be either positive or negative. It will move up to but not past either the beginning or the end of the buffer.

2.5.4 Status and Complex Movement

UNSIGNED
BGetCol()

Returns the display width of the text between the beginning of the buffer or the latest Newline and the Point. Normally, this will be the column that the cursor is displayed in.

BMakeCol(column)

int column

Moves the Point so that the display width of the text between the beginning of the buffer or the latest Newline and the Point is as near Column as possible. The Point is placed at the end of the current line if Column is beyond the end. If Column falls in the middle of a multi-column character, the Point is placed after the character.

UNSIGNED

BLocation()

Returns the Position of the Point in the buffer in units of characters from the beginning of the buffer. This value is not correct for locations above 65,535.

UNSIGNED

BLength(buffer_descriptor)
int buffer_descriptor

Returns the length of the buffer indicated by Buffer_descriptor in characters. This value is not correct for buffers of more than 65,535 characters.

FLAG

BCSearch(what)
char what

Search forward through the buffer starting at the Point for character What. Returns TRUE if the character was found, FALSE otherwise. The Point is left after the character that was searched for if it was found or at the the end of the buffer if it was not.

FLAG

BCRSearch(what)
char what

Search backward through the buffer starting at the Point for character What. Returns TRUE if the character

was found, FALSE otherwise. The Point is left before the character that was searched for if it was found or at the beginning of the buffer if it was not.

FLAG

BModp(buffer descriptor)
int buffer_descriptor

Returns TRUE if the buffer indicated by Buffer_descriptor has been modified since it was created or had a file operation (e.g. read or write) performed upon it, FALSE otherwise.

char
Buff()

Returns the character after the Point. Results are undefined if the Point is at the end of the buffer.

2.5.5 Mark Manipulation

int
BCreMrk()

Creates and returns a mark and places it at the Point. An error message is given and NULL is returned if there are no more available marks.

BKillMrk(amark)
int amark

Removes the mark Amark.

BMrkToPnt(amark)
int amark

Places the mark Amark at the Point.

BPntToMrk(amark)
int amark

Places the Point at the mark Amark. The passed mark must be in the current buffer and an error message is given if it is not.

BSwapPnt(amark)
int amark

Interchanges the positions of the Point and the mark Amark. The passed mark must be in the current buffer and an error message is given if it is not.

FLAG
BIsAtMrk(amark)
int amark

Returns TRUE if the positions of the Point and the mark Amark are the same, FALSE otherwise. The passed mark must be in the current buffer and an error message is given if it is not.

FLAGB
IsBeforeMrk(amark)
int amark

Returns TRUE if the position of the Point is before the position of the mark Amark, FALSE otherwise. The passed mark must be in the current buffer and a value of FALSE is returned and an error message is given if it is not.

FLAG
BIsAfterMrk(amark)
int amark

Returns TRUE if the position of the Point is after

the position of the mark `Amark`, `FALSE` otherwise. The passed mark must be in the current buffer and a value of `FALSE` is returned and an error message is given if it is not.

Screen marks are special marks used by `redisplay` to improve its performance. Each mark has a flag associated with it. `Redisplay` clears all of these flags and the buffer modification routines (e.g., `BInsert` and `BDelete`) set the flag on the the last mark located before the modification. As the `redisplay` places the marks on the beginning of each screen line, the flag serves to indicate whether that line has changed since the last `redisplay`.

```
int
BScrnMrk(indx)
    int indx
```

Returns the screen mark corresponding to the `Index`'th row of the screen.

```
FLAG
BTstMrk(smark)
    int smark
```

Returns the state of the mark `Smark` and resets the state to `FALSE`. It is used internally by `redisplay`. Results are undefined if it is invoked.

```
BSetMod(smark)
    int smark
```

Sets the state of the mark `Smark` to `TRUE`. This will result in the corresponding line being `redisplayed` the next time `IncrDsp` is invoked.

2.5.6 Reading and Writing Files

```
FLAG
```

BReadFile(filename)
char *filename

Read the contents of the file Filename into the buffer. The current contents of the buffer are lost. All marks and the Point are placed at the beginning of the buffer. Succeeded is TRUE if the read was successful, FALSE otherwise. The buffer's modified flag is cleared.

BWriteFile(filename)
char *filename

Write the contents of the buffer into the file Flename. The buffer's modified flag is cleared.

BFlush()

If there are any modified pages, one is written to the swap file, otherwise, nothing happens.

2.5.7 Private Routines

The following are routines private to the buffer abstraction. They may not be invoked from outside the buffer abstraction.

DskWarn	DskUnWarn	free_page
get_memp	GetGap	into_mem
make_cur	make_offset	new_page
page_split	SetMod	SubSet

Chapter 3

Source Code

Note that not all routines will be documented here; many of the commands are left out. Understanding these more simple routines is tantamount to understanding what the command is supposed to do.

3.1 Control Commands: File COMM1.C

MArg()

Does the C-U (Universal Argument) command.

Tmp accumulates the numeric argument as typed in (e.g. '5' then '3' becomes the value fifty-three). Tmp does NOT accumulate the multiplications by four. Cflag tells whether or not a digit or digits was entered. Eflag tells whether the argument has been echoed and thus needs clearing.

As the routine is entered, any previous argument is multiplied by four. A (possibly null) digit sequence is picked up and accumulated. If the digit sequence was non-null, the old argument is thrown out and those digits become the argument. You can thus end an argument with any number of C-U's. The non-digit which ends the digit sequence then gets dispatched as a command. If a C-U was picked up, this routine is invoked recursively.

3.2 Meta Commands: File COMM2.C**MDeleLine**

Does the M-C-K (Kill Entire Line) command.

The tricky thing in this routine is that it does the delete in two separate operations. It first deletes the text from the Point to the beginning of the line and puts it at the START of the delete buffer. It then deletes the text to the end of the line (including the Newline) and puts it at the END of the delete buffer. If the command is given an argument, the Point is left at the beginning of the following line and reinvoked. All of the text from that and following lines will be put at the end of the delete buffer.

MCapWord

Does the M-C (Capitalize Word) command.

The tricky thing here is that if the word that is being capitalized is only one character long, you don't want to call the lowercase word routine.

MFillPara

Does the M-Q (Fill Paragraph) command.

It begins by resetting the fillwidth if there was an argument. It then creates a different mark so that it can return there when it is done.

The rest of the initialization involves putting the point and a mark in the right places. A BMove(-1) is done so that it will fill "this" paragraph if the Point is just after the last

non-white character (to wit, the final period of the final sentence). It then goes to the end of the paragraph and backward a character yet again. If you are at the end of the buffer (BIsEnd is TRUE), you must have a null buffer and so return. If that character is whitespace, you have only whitespace in that direction in your buffer (Forward Paragraph always leaves you just after non-whitespace, if possible), therefore, you return. Otherwise, you set a mark there and go back to the beginning of the paragraph to start filling.

The basic flow of the rest of the command is to bounce through the paragraph, stopping at each Newline and before each word and doing some processing, finally finishing when you reach that mark at the end of the paragraph. Note that breaks are only made on whitespace (e.g., it won't split "a-b") and whitespace can be deleted or inserted by the command.

There are two cases. First, a word can end after Fillwidth. Second, a Newline can occur before Fillwidth. In the first case, you jump backwards to the previous whitespace, delete it, insert a Newline and any indentation, and jump forward again. You never have to back up past whitespace more than once (unlike MFillChk, the auto fill space routine) because you were just there and it wasn't past Fillwidth. You jump forward to whitespace as you finish for efficiency and to keep from hanging in an infinite loop if you have a single word longer than Fillwidth. In the second case, you delete the Newline and any indentation and insert a space.

In the center of the text of the routine and separating the two cases is the mechanism for switching between the two cases. No matter which case you just processed, you skip over any blanks or Tabs and stop when you get to a Newline or a non-whitespace character. A call to IsNL is made to see which case it was that you encountered and the result of that call selects the case to handle.

The routine finishes by returning the Point to the original mark and cleaning up.

Note that in this scheme all Newlines will be deleted and re-inserted, even though no actual motion of text is needed. *sigh*.

MCntrLine

Does the M-S (Center Line) command.

This begins by getting Fillwidth and Arg to be the same value. Arg performs double duty here. It comes in as the argument, of course, but it goes on to become the value Fillwidth - Indentcol.

The routine itself moves to the beginning of the line, deletes any whitespace (for example, the stuff it put in last time if the line is being re-centered), goes to the end of the line, gets its position, goes back to the beginning, inserts the right number of spaces, and leaves you at the end.

MFillChk

Does the Fill Mode Space (Auto Fill Space) command.

It begins by returning if it doesn't have to do anything (the Point is before Fillwidth). If the Point is at or after Fillwidth, it has to split the line. It jumps back to whitespace until it gets before Fillwidth (it can't assume that the immediately previous whitespace is before Fillwidth). It then deletes that whitespace.

Now things get hairy. As you might have noticed, it placed a mark before it did anything. If it is still at that mark, that means there was lots of trailing whitespace on the line and it is the whitespace that put you over Fillwidth. It sets a flag indicating what the situation was. It then inserts the Newline and does the indenting. If that flag indicates the this is a reasonable situation (it was a word that put you over), it puts you back at the mark (which should be after

the word that was wrapped) and inserts a space. If it was the spaces that put you over, it doesn't have to insert the space as the Newline and indentation already is the space.

3.3 Control-X Commands: File COMM3.C

MLstBufs

Does the C-X C-B (List Buffers) command.

Note how it uses BSetMod and BScrnMrks to tell the redisplay what lines it bashed. Note also that it waits for the next character to be typed in before it allows the screen to be cleared.

MFindFile

Does the C-X C-F (Find File) command.

Yes, there is a call to BReadFile lurking in there. This command is shorthand for three different things and it shows, both in the complexity of the documentation and in the complexity of the code.

MDelMode

Does the C-X C-M (Delete Mode) command.

This routine, as with Add Mode, assumes that the mode list is kept packed and stored from the top down (large indices to small indices) in Buffs.Bmodes. Thus, [0 0 ModeA ModeB] is legal, while [0 ModeA 0 ModeB] and [ModeA ModeB 0 0] are not. These examples are number the indices from zero to three, from left to right.

Given that convention, Delete Mode's job is easy. It scans down the mode list until it finds a match, then packs the rest of the list.

3.4 Support Routines: File SUPPORT.C

FLAG

ArgEcho(targ)
int targ

Uses the global cnt.

Waits an amount of time proportional to DELAY. If a character has been entered, returns FALSE. If not, prints "Arg: " and the argument Targ, and returns.

FLAG

Ask(mesg)
char *mesg

Prints the message in the echo line. If the user responds with "Y", "y", or " ", returns TRUE. If the user responds with "N", "n", DEL (^?), or BS (^H) returns FALSE. Otherwise, it rings the bell and checks another character.

BlockMove(from,to)
int from, to

Moves the (possibly zero length) block of characters between the Point and the mark From to the mark To. To is left after the moved characters. The Point is left at mark From. The Point is assumed to be before the mark From. If it is after, no characters are moved.


```
change(old,new)
    char *old, *new
```

Assumes that the Point is at the end of Old. It deletes Old and inserts New.

```
int
CheckMode(tmname)
    char *tmname
```

Returns the modeid of the indicated mode if tmname is a mode name, otherwise FALSE. The mode identifier is typically the first character of the mode name, e.g. "f" for Fill mode.

```
ClrEcho()
```

Clears the echo line.

```
CopyToMrk(tmark,forwdp)
    int tmark, forwdp
```

Imports the globals del_buff and lfunct.

Copies the region (the text between the Point and Tmark) into the delete buffer. If Forwdp is TRUE, it appends the text to the end of the delete buffer. Otherwise, it puts it at the beginning. It calls DelCmd and if TRUE is returned, it saves what is in the delete buffer, otherwise the region replaces what is already there.

```
FLAG
Debug(message,value)
    char *message
    int value
```

Prints Message and Value in the echo area, calls redisplay to allow you to see what the buffer looks like, and waits for a character. Returns

TRUE if the character is DEL (^?) or BS (^H),
FALSE otherwise.

FLAG

DelayPrompt(mesg)
char *mesg

Uses the global cnt.

Waits for DELAY. If a character has been entered, returns FALSE. If not, prints the message in the echo line and returns TRUE. This routine is used to accomplish the delayed echoing of "Meta:", etc.

FLAG

DelCmdnd(lfunct)
int (*lfunct)()

Imports the global lfunct.

Returns TRUE if Lfunct points to a command that saves deleted text in the kill buffer, FALSE otherwise.

DoReplace(query)
int query

Uses the global tmark.

Does the Replace String and Query Replace commands. Query is a flag that tells it whether or not to ask each time it locates an occurrence.

The routine is straightforward except in two cases. First, all characters not specifically checked for (the default) act as "No" answers. This is implemented in the code by not executing the remainder of the while loop because it executed the continue statement.

The other sticky point is in handling the ',,' (Replace And Confirm) command. First, if it is a

query replace, the routine sets the Tchar variable to ',' in order to cause the "Replacing" message to be printed. Each time ',' command is entered, you will be asked to confirm the replace and this message will have been bashed. Therefore, it is refreshed the next time through the loop and that same mechanism is used to get the whole mess going. The ',' command does the replace, so in that respect it is treated just like "Yes." However, after the change has been made, it must check to see if the command was ',' and, if so, ask for confirmation.

Note that the ',' option does not save the original string that was being replaced. Thus, if the "put it back" option is selected, the upper/lower caseness of the original string is lost.

Echo(mesg)
char *mesg

Prints the message in the echo line.

error(mesg)
char *mesg

Displays an error message. An error message appears off to the right in the echo line. Waits for a character to be typed before returning.

ForceCol(col,forwardp)
int col, forwardp

This routine will force the Point to be in column Col. If necessary, it will insert spaces to put you there. Forwardp tells whether to round up or down on multi-column characters. (Due to multi-column characters, you are not always going to be in column Col. ForceCol merely deals with the interesting cases of Newline and Tab characters preventing you from being in the desired column.)

First, don't touch this routine. If you change anything, it probably won't work. This routine also relies heavily on the behavior of BMakeCol. BMakeCol leaves you in the proper column, if possible. If the proper column falls in the middle of a multi-column character (e.g., Tab or ^A), it leaves you after the character. If the line isn't long enough, it leaves you at the Newline.

ForceCol begins by doing a BMakeCol. That ensures that you are at least close to the desired position. It then checks to see if the desired column is less than or equal to zero. If it is less than, return. If it is equal to zero, we know that BMakeCol succeeded in leaving us in the correct column.

There are now three cases. First, Col can be after the Newline. It is detected by BGetCol < Col. In this case we want to insert some spaces. The second case is for Col to have fallen in the middle of what a Tab is tabbing over. It is detected by BGetCol > Col and the character before the Point is a Tab. In this case, it moves back one character and inserts some spaces. The third case is where Col is in the middle of a multi-column character (not Tab, though). In this case, it leaves you before or after the character depending upon the sense of Forwardp. There is an implicit fourth case, that in which you are already at the correct column. In this case, nothing must be done.

The purpose of the big hairy if statement on the third line is to leave you before the Tab if there is one (the second case above). The Indent on the following line then can handle both the first and second cases at one fell swoop. Note that if Indent is given a negative or zero argument, it does nothing. Thus, if you are already at the correct column or you are beyond it (third case), it does nothing.

The last if statement handles the third case. All other cases will have BGetCol == Col by now. All it does is leave you on the indicated side of a multi-column character.

FLAG

```
GetArg(mesg, term, str, len)
char *mesg, term, *str
int len
```

This routine does all of the work for accumulating a string argument (e.g., a search string or file name). It provides the full echoing and line editing facilities needed.

It prints the message and accumulates the response into Str. Str can have up to Len characters. When Term is typed, the routine returns. It returns TRUE if everything went all right, FALSE if C-G (Abort/Cancel Prefix) was entered.

int

```
GetModeId(msg)
char *msg
```

Calls GetArg with Msg as the prompt and returns a Modeid of a valid mode or prints an error message and returns FALSE if a valid mode was not entered.

```
index(tstr, tchar)
char *tstr, tchar
```

Returns the index of the first occurrence of Tchar in Tstr, or -1 if there is no occurrence.

FLAG

```
IsClose()
```

Returns TRUE if the character after the Point is a "closing" character, FALSE otherwise. A closing character is one of),], }, ", or '. Used by the sentence movement commands.

FLAG
IsGray()

Returns TRUE if the character after the Point is a "gray" character, FALSE otherwise. A gray character is a Space, a Tab, or a Newline. Note that IsGray includes Newlines, while IsWhite does not.

FLAG
IsNL()

Returns TRUE if the character after the Point is a Newline, FALSE otherwise.

FLAG
IsNLPunct()

Returns TRUE if the character after the Point is a Newline or punctuation, FALSE otherwise. Punctuation is either ".", "?", or "!".

FLAG
IsParaEnd()

Returns TRUE if the character after the Point is a Newline, Tab, "", or ".", FALSE otherwise. It is intended to be used to determine whether the Point is at the end of a paragraph. Paragraphs are delimited by Newline Newline, Newline Tab, Newline "" (Scribble commands), or Newline "." (most other text formatter commands). This routine assumes that you are between the two delimiters and it only checks the second one.

FLAG
IsSentEnd()

This one is a monster. You are assumed to be just after a candidate for an end of sentence (to

wit, ".", "!", or "?"). This routine moves you over an arbitrary number of){}'" characters and stops at the first character that isn't one of those. If that character is a grayspace character, it returns TRUE, otherwise it returns FALSE. Note that this will leave you at the grayspace or the other character.

FLAG
IsToken()

Returns TRUE if the character after the Point is a token character, FALSE otherwise. Token characters are alphabetic and digits.

FLAG
IsWhite()

Returns TRUE if the character after the Point is a whitespace character, FALSE otherwise. A whitespace character is either a Tab or a Space. Note that IsWhite does not include Newlines, but IsGray does.

itot(n)
unsigned n

Prints N on the terminal.

KbWait()

Waits for a character to be typed. It writes out modified pages (by calling BFlush) so long as no character has been typed.

KillToMrk(tmark, forwdp)
int tmark, forwdp

Deletes the region (between the Point and Tmark)

and saves the deleted text in the delete buffer. Forwdp tells whether to put the deleted text at the beginning or the end of the delete buffer.

lowcase(str)
char *str

Converts the string to lower case.

ModeFlags()

Uses the globals buffs, cbuff, and lfunct.

Prints the percentage, modified, and append next delete flags on the mode line.

MovePast(pred, forwdp)
int (*pred)(), forwdp

Moves through the buffer, invoking Pred at each character. It stops when Pred returns FALSE, leaving the Point on the near side of the character which caused Pred to return FALSE. Forwdp tells whether to move forward or backward. Pred is a pointer to a function of no arguments.

MoveTo(pred, forwdp)
int (*pred)(), forwdp

Moves through the buffer, invoking Pred at each character. It stops when Pred returns TRUE, leaving the Point on the near side of the character which caused Pred to return TRUE. Forwdp tells whether to move forward or backward. Pred is a pointer to a function of no arguments.

NLPrnt(str)
char *str

Print Str at the terminal. Newline characters are printed as "<NL>".

FLAG
NLSrch()

Puts you after the next Newline or at the end of the buffer if there isn't one in that direction. Returns TRUE if it found one, FALSE otherwise.

Rebind(from,to)
int (*from)(), (*to)()

Updates the global functs.

Changes all occurrences of the the function From in the bindings tables to To.

FLAG
RNLSrch()

Puts you before the previous Newline or at the beginning of the buffer if there isn't one in that direction. Returns TRUE if it found one. FALSE otherwise.

RubOut(ostr,str,tcol)
char *str, *ostr, tcol

Performs DEL hackery for GetArg. It handles deleting multi-column characters from the screen.

SIndent(arg)
int arg

Uses the global cnt.

Inserts Arg spaces. Does nothing if Arg is negative or zero.

```
strip(to,from)
  char *to, *from
```

From is a file name. This routine strips off the device ("a:") if it exists and the extension (.doc) if it exists and returns what's left in To. This routine does not change the case of anything.

FLAG

```
StrSrch(str,forwardp)
  char *str
  int forwardp
```

Uses the global cnt.

Does a string search in the direction indicated by Forwardp. Returns TRUE if it found the string, FALSE otherwise. Leaves you after the string or at the end of the buffer if it is a forward search, before the string or at the beginning of the buffer if it is a backward search.

```
ToBegLine()
```

Moves you to the beginning of the current line.

```
ToEndLine()
```

Moves you to the end of the current line.

```
TIndent(arg)
  int arg
```

Indents Arg columns with Tabs and spaces. It assumes that the Point was in column zero (or at least at a Tab stop).

ToNotWhite(forwardp)
int forwardp

Moves you to the first non-whitespace in the direction indicated by Forwardp. You are not moved if you are already at non-whitespace.

ToSentEnd(forwardp)
int forwardp

Moves you to the end of a sentence in the direction indicated by Forwardp. This routine does the work for Backward Sentence and Forward Sentence.

It first finds a potential sentence end (a punctuation mark or Newline). If the candidate is a Newline, it sees whether or not it is the end of a paragraph. If the candidate is a punctuation mark, it places a mark there, calls IsSentEnd (which moves you somewhere), and restores your position. It then checks to see if you are done.

ToWhite(forwardp)
int forwardp

Moves you to the first whitespace in the direction indicated by Forwardp. You are not moved if you are already at whitespace.

ToWord()

Moves you forward to the beginning of a token (word). You are not moved if you are already at a word.

upcase(str)
char *str

Converts Str to upper case.

FLAG

VMovCmnd(lfunct)

int (*lfunct)()

Returns TRUE if the previous command was a vertical movement command, FALSE otherwise.

Chapter 4

The Terminal Abstraction

The terminal data abstraction is responsible for handling all of the interaction with the user's console. It provides a uniform interface to any terminal that Mince would ever see. This interface standardizes the calls for performing operations such as cursor positioning and clearing parts of the screen. It also provides for displaying characters in a uniform manner across all terminals.

The terminal abstraction performs these tasks by defining a "virtual terminal." By virtual terminal, we mean that the interface will always perform the desired function; it is up to the abstraction to make up for any missing or unusual features. It is the responsibility of the terminal abstraction to ensure that the characteristics of this virtual terminal are faithfully reproduced on any physical terminal.

The virtual terminal has a screen that is a two dimensional array of characters. They are numbered with (0,0) in the upper left corner and (TMaxCol()-1,TMaxRow()-1) in the lower right corner of the screen. There are both a screen point and a cursor. The screen point is the (x,y) coordinate where the next modification to the screen will take place. The cursor is the (x,y) location where the visual marker is displayed. Note that these are not necessarily the same place: the screen point can be moved quite a bit while the cursor stays in the same place. (On most terminals, modifications must take place at the cursor. Thus, when an actual change is being made to the screen, the cursor must first be moved to that place.) The virtual terminal does not know about its own right edge. Thus, strange things can happen if, for example, a multi-column character is displayed so that it might wrap. (Mince's redisplay avoids this case.)

The keyboard for this virtual terminal has a buffer,

typically eighty characters or so long. The terminal will only remember these characters (i.e., implement typeahead) if it is checked often enough. Thus, those calls to `TKbChk` which are liberally interspersed throughout `Mince` (and should be included in any changes that you make) perform a vital task.

The terminal abstraction is tailored for a specific terminal by running `Config`. `Config` stores the description in the swap file, which is in turn read by `Mince` as `Mince` comes up. A description of the place this information is read into and the format of the file can be found in Chapter One.

4.1 Initialization and Termination Routines

These globals are used throughout the terminal abstraction.

Private globals `prow`, `pcol`, `srow`, `scol`, `clrcol`, and `kbdq`.

The following routines initialize and terminate the terminal abstraction.

`TInit()`

Initialize the terminal abstraction and the terminal.

The routine first initializes the keyboard queue (buffer). It then sends the initialization string (as defined in the configuration program), sets the `Clrcol` array to the last column, and clears the screen (which sets them again).

`TFin()`

Restore the terminal to its original state.

Forces the cursor to be displayed where it "belongs" (at the bottom of the screen) and

deinitializes the terminal by sending the string defined in the configuration program.

4.2 Cursor Positioning

int
TGetRow()

Returns the row that the screen point is on.

int
TGetCol()

Returns the column that the screen point is on.

int
TMaxRow()

Returns the number of rows on the screen. Row-1 is the number of the last row on the screen.

int
TMaxCol()

Returns the number of columns on the screen. Column-1 is the number of the last column on the screen.

TSetPoint(irow,icol)
int irow, icol

Sets the screen point to (irow,icol).

TForce()

Forces the visual cursor to be displayed at the screen point.

Does nothing if the cursor is already there. If not, it uses the cursor positioning sequence to put it there.

4.3 Display Routines

TBell()

Rings the terminal bell or performs other alarm indications by sending the bell string.

TCLEOL()

Clears from the screen point to the end of the line.

Sends the clear to end of line string if there is one, otherwise it sends the correct number of blanks. It pays attention to and sets clrcol.

TClrLine()

Clears the line that the screen point is on. The screen point is left at the beginning of the line.

TCLEOW()

Clears from the screen point to the end of the screen.

Sends the clear to end of window string if there is one, otherwise it calls TCLEOL, then repeats going to the next row, setting the screen point column to zero, and calling TCLEOL until it gets to the end of the screen. It then restores the screen point to where it was.

TClrWind()

Clears the entire screen (window). The screen point is left at the beginning (home position) of the screen.

Sets the screen point to (0,0). Sends the clear window string if one exists, otherwise calls TCLEOW.

4.4 Printing Text

TPrintChar(ichar) char ichar

Imports the global tabincr.

Prints the character at the screen point and updates the screen point by the display length of the character. Ordinary characters are printed normally. Control characters are printed as "^" followed by the character Ichar XOR 64 (e.g., C-A prints as ^A). Meta characters are printed as "~" followed by the character Ichar AND 127 (which may be a control character). Tabs (^I) are printed as the number of spaces remaining before the next tab stop (determined by Tabincr).

On ordinary characters, it puts the cursor at the screen point, sends the character, updates the cursor and screen point columns, and updates Clrcol.

On other characters, it prints them "in pieces" by calling itself recursively or, in the case of Newline, implements the meaning of the character by logically printing a Carriage Return and a Line Feed.

```
TPrintStr(string)
  char *string
```

Prints the string as if by repeated calls to TPrintChar.

```
TDisStr(row,col,string)
  int row, col
  char *string
```

Sets the screen point to (row,col) and prints the string as if by TPrintStr.

```
int
TWidth(colcnt,tchar)
  int colcnt
  char tchar
```

Imports the global tabincr.

Returns the display width of the character as it would be printed by TPrintChar with the screen point in column colcnt. If the character should be wrapped to the next line, the extra spaces needed to display the wrapping are included in Width. Note that the display width of a Newline is the negative of colcnt.

4.5 Low Level Output and Keyboard Drivers

TPutChar(ochar)
char ochar

Sends a character to the terminal without any interpretation.

It actually does the output. It uses the bios if possible (based on the global outport.biosp). If it can't, it does the output manually, waiting for the port to be ready and then sending the character.

TKbChk()

Checks to see if there is a character available from the keyboard. If there is, it reads the character in and places the character in a FIFO queue. This routine should be called frequently in order to maintain proper typeahead buffering. If the queue is full, it rings the terminal bell.

It actually does the input. It checks the status, using the bios if possible. If there is a character ready, it reads it in, again using the bios if possible.

FLAG
TKbRdy()

Returns TRUE if there is a character available, FALSE otherwise.

It calls TKbChk, then checks the queue to see if there is a character ready.

char
TGetKb()

Returns the next available character. It will wait if necessary for a character to become available.

Waits in a loop, calling TKbChk until there is a

character ready, then grabs it.

4.6 Internal Routines

put_string(sdef)
 struct str *sdef

Puts a command string (e.g., the clear to end of line string) which is represented as a structure in the terminal abstraction internal format.

put_coord(firstp)
 int firstp

Sends a cursor coordinate to the terminal. The argument, Firstp, indicates WHICH coordinate to send; the coordinates themselves are stored as globals (Prow and Pcol). If Firstp is the same sense as Terminal.Rowfirstp (i.e., they are both TRUE or both FALSE), the row is sent, otherwise, the column is sent. In either case, the coordinate is biased as necessary (Terminal.Rowbias or Terminal.Colbias). If the coordinate is sent in binary, it will optionally complement it (Terminal.Compp).

put_num(num)
 unsigned num

Sends a number to the terminal as ASCII digits.

If the number is greater than ten, it calls itself recursively to print out the first n-1 digits. It then prints the nth digit. Note that it calls TPutChar while itot (in SUPPORT.C) which does a similar thing calls TPrntChr. More often than not, this routine is called by TPrntChr in the course of doing a cursor positioning sequence.

(Command level routines do not call this routine--they call itot.)

```
put_pad(npads)
  int npads
```

Does the padding as appropriate, either by sending Npads padding characters or by counting Npads times.



About This Chapter

This chapter was originally written and submitted as a B.S. thesis at the Massachusetts Institute of Technology. Its goal was to provide a discussion of considerations of implementing a text editor.

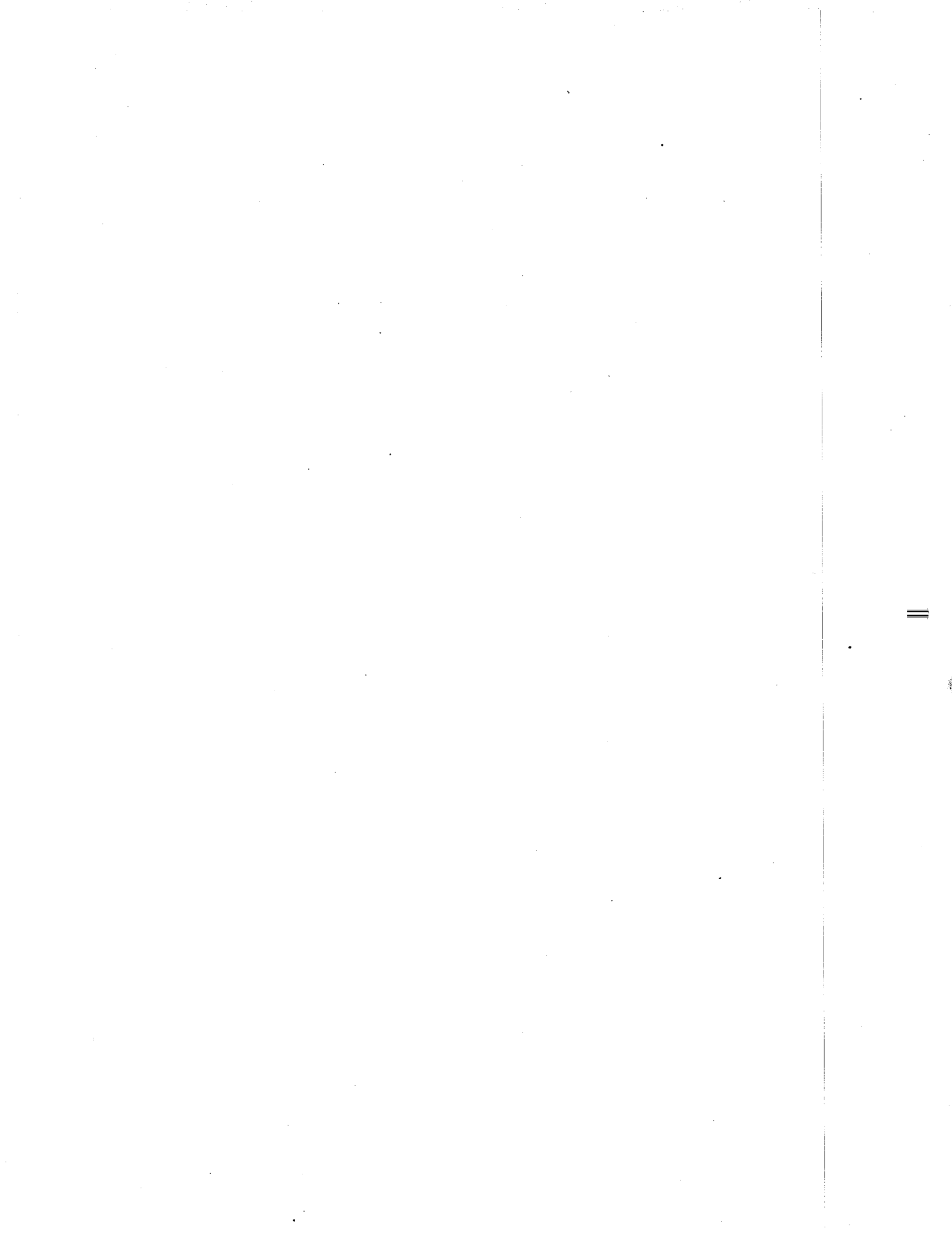
The preceding chapters of the Mince Internal Documentation discuss the details of a specific implementation; this chapter will help provide perspective about what considerations are general and what ones are implementation specific.

Acknowledgements

I would like to thank Owen Ted Anderson for teaching me a lot of what I know about editors as well as writing one of the most readable programs around.

I would like to thank Bernard S. Greenberg, who supplied some of the algorithms which are presented here.

I would like to thank Richard M. Stallman and the rest of the M.I.T. Artificial Intelligence Laboratory for creating the original Emacs and for doing most of the development of ITS EMACS.



1. Introduction

This thesis is intended to answer the question, "What are the important considerations in designing a text editor?" In answering this question, it will provide a reference document for would-be implementors of text editors.

There is a modest amount of literature available which discusses topics related to text editing. Most of the papers are "reference manual"-like because they explain the user interface only. A few of the rest cover the details of a specific implementation of an editor. This thesis will generalize the latter into a document which considers the problems relevant for all text editors.

The primary goal of a text editor is to allow the user to edit text. There are two secondary goals. First is to perform this editing without wasting resources. Second is to give the user a pleasant environment to edit in. The latter requires a good command set, feedback to the users, and quick response to commands.

Achieving these goals is hard. One way to make it easier is to break the design of the editor into three parts. The memory management part performs efficient editing of the text. It is essentially a very simple editor in itself. The incremental redisplay part provides feedback to the user. The command set (loop) part translates the user's input into commands to the memory management part. Each part of the structure contributes in its own way towards providing quick response. It is this structure that will be discussed in this thesis. Each chapter of the thesis covers a different part.

The second chapter is memory management (you are reading the first chapter). The basic problem that is addressed is: given that you have a possibly large buffer, how do you structure the storage for it so that trivial operations (e.g., inserting a character and moving around in the buffer) do not require excessive amounts of work? Other problems are: what should the interface to the buffer look like from a program? How do these considerations change when you have multiple buffers and/or virtual memory? In a nutshell, this chapter discusses the cpu time - memory - disk channel tradeoff. This topic is interrelated with the next one.

The third chapter is incremental redisplay. The basic problem here is: given that the user has a reasonable video terminal which you can communicate with over a limited bandwidth channel, how do you change what is displayed on his screen to match the current contents of the buffer? Other problems are: what are reasonable terminals to use? What extra information can you retain to speed up the updating process? This chapter discusses the cpu time - I/O channel usage tradeoff.

The fourth chapter is a discussion of the command loop. What is the basic edit cycle? What sort of errors do you have to recover from? How and why do you dynamically change the editor itself? What are some criteria to use when selecting an implementation language?

The fifth chapter considers user interface hardware. What are desirable ways for the user to interact with the editor? This area includes such things as desirable features in keyboards and how to take advantage of graphical input.

The sixth chapter mentions some other uses for text editors.

Note that MIT Emacs will be used in this thesis whenever a reference to a specific editor is required (for example, when discussing command syntax). This class of editors will be referred to as

"Emacs-type." A specific editor was selected (as opposed to creating another one) specifically to avoid the work of reinventing the wheel. MIT Emacs was selected because of the author's familiarity with it and because several implementations of it have been made, thus providing a wealth of experience with it in different environments.

Addendum for Mince Version 2.6

There have been no functional changes from Version 2.5. However, portions of Mince have been coded in assembly language to improve performance. These changes are nearly transparent to anyone writing and compiling Mince code, but the link phase is different. The new link line is:

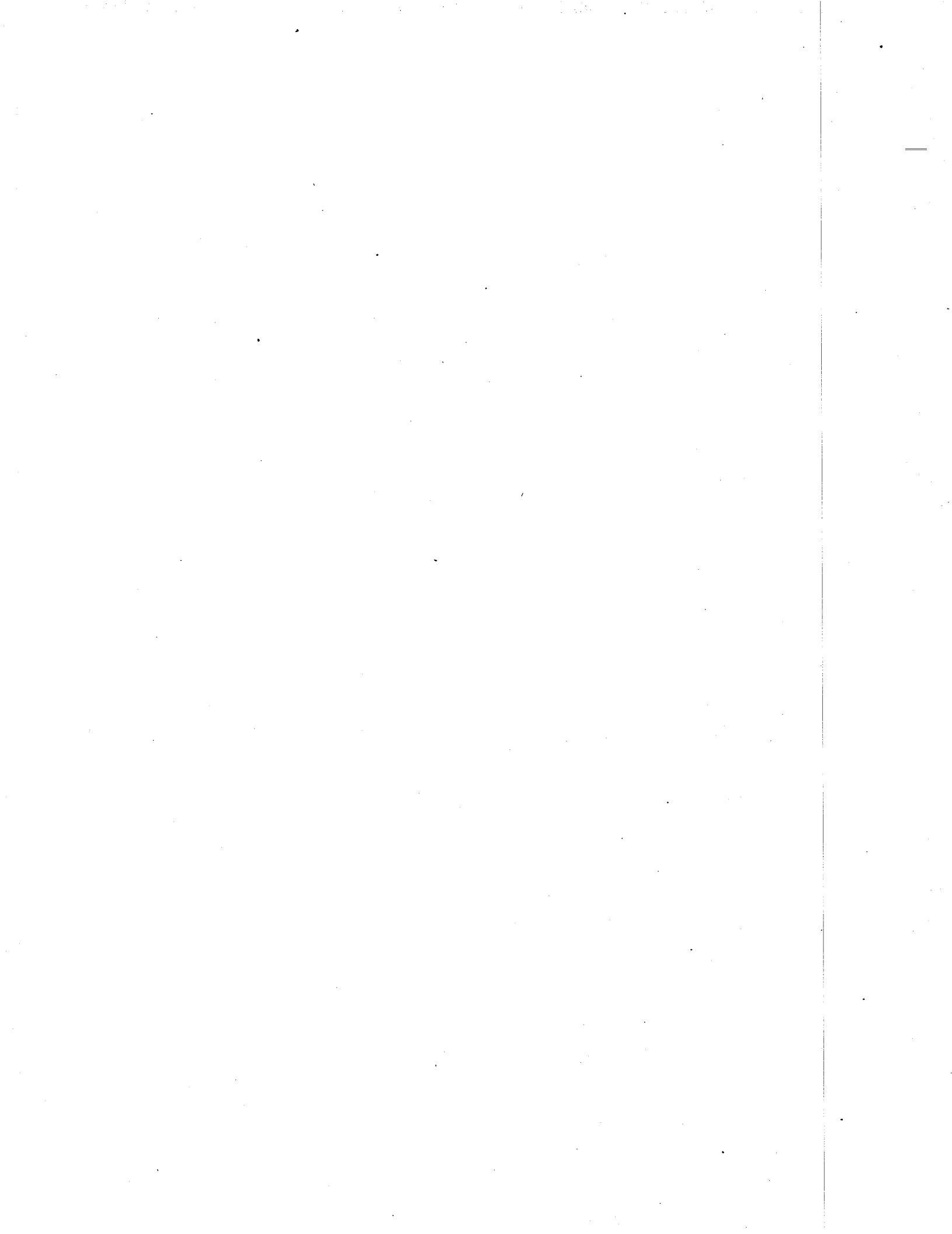
```
l2 mince bindings comm1 comm2 comm3 vbuff1
  vbuff3 vbuff2 aterm -l support term util
```

If you don't alter the terminal abstraction at all, the preceding link line is the one to use. A fraction of the terminal abstraction is written in assembler. Thus, if you do alter the terminal abstraction and find yourself dealing with those portions, you have two choices.

1. You can write the changes in assembler. In this case, edit ATERM.ASM, use MAC to assemble it, and use the above link line.
2. You can write the changes in C. In this case, edit TERM.C, use the BDS C compiler to compile the changes, and use the following link line:

```
l2 mince bindings comm1 comm2 comm3 vbuff1
  vbuff3 vbuff2 cterm -l support term util
```

Refer to the source to ATERM.ASM and TERM.C for more detailed commentary.



Mince Commands

All printing characters: a-z, A-Z, 0-9, space, and
!"#\$%&'()*+,-./:;<=>?@[]^_{|}~ Self-insert

<LF> see C-J
<CR> Newline Insert
<TAB> Tab Insert
 Delete Character Backward
<ESC> Meta-command Prefix

C-@ Set Mark
C-A Beginning of Line
C-B Backward Character
C-D Delete Character Forward
C-E End of Line
C-F Forward Character
C-G Abort/Cancel Prefix
C-H see
C-I see <TAB>
C-J Newline and Indent
C-K Kill Line
C-L Redisplay Screen
C-M see <CR>
C-N Next Line
C-O Open Line
C-P Previous Line
C-Q Quote Next Character
C-R Reverse String Search
C-S Forward String Search
C-T Transpose Characters
C-U Universal Argument
C-V View Next Screen
C-W Wipe Region
C-X C-X Command Prefix
C-Y Yank Killed Text
C-[see <ESC>
C-\ Delete Indentation

C-X <TAB> Set Tab Spacing
C-X C-B List Buffers
C-X C-C Exit to Command Level
C-X C-F Find File
C-X C-I (same as C-X <TAB>)
C-X C-M Delete Mode

C-X C-R Read File
C-X C-S Save File
C-X C-V View Next Screen Other Window
C-X C-W Write File
C-X C-X Exchange Point and Mark
C-X C-Z View Previous Screen Other Window
C-X . Set Indent Column
C-X = Where Am I
C-X 1 One Window
C-X 2 Two Windows
C-X B Select Buffer
C-X F Set Fill Column
C-X K Kill Buffer
C-X M Add Mode
C-X O Other Window
C-X ^ Grow Window

M- Delete Word Backward
M-<SPACE> see C-@
M-< Beginning of Buffer
M-> End of Buffer
M-A Backward Sentence
M-B Backward Word
M-C Capitalize Word
M-D Delete Word Forward
M-E Forward Sentence
M-F Forward Word
M-H Mark Whole Paragraph
M-K Kill Sentence Forward
M-L Lowercase Word
M-Q Fill Paragraph
M-R Replace String
M-S Center Line
M-T Transpose Words
M-U Uppercase Word
M-V View Previous Screen
M-W Copy Region
M-[Backward Paragraph
M-\ Delete Surrounding Whitespace
M-] Forward Paragraph

M-C-H see M-
M-C-K Kill Entire Line
M-C-R Query Replace String
M-C-W Append to Kill Buffer

Fill Mode Commands

space Auto Fill Space

Page Mode Commands

All printing characters: a-z, A-Z, 0-9, space, and

!"#\$%&'()*+,-./:;<=>?@[]^_{|}~ self-overwrite

C-A To First Non-White
C-B Backward Character on Line
C-E To Last Non-White
C-F Forward Character on Line
C-H Overwrite Character Backward
C-N Next Line Forced
C-P Previous Line Forced
C-Q Quote Next
C-X \ Delete Trailing Whitespace



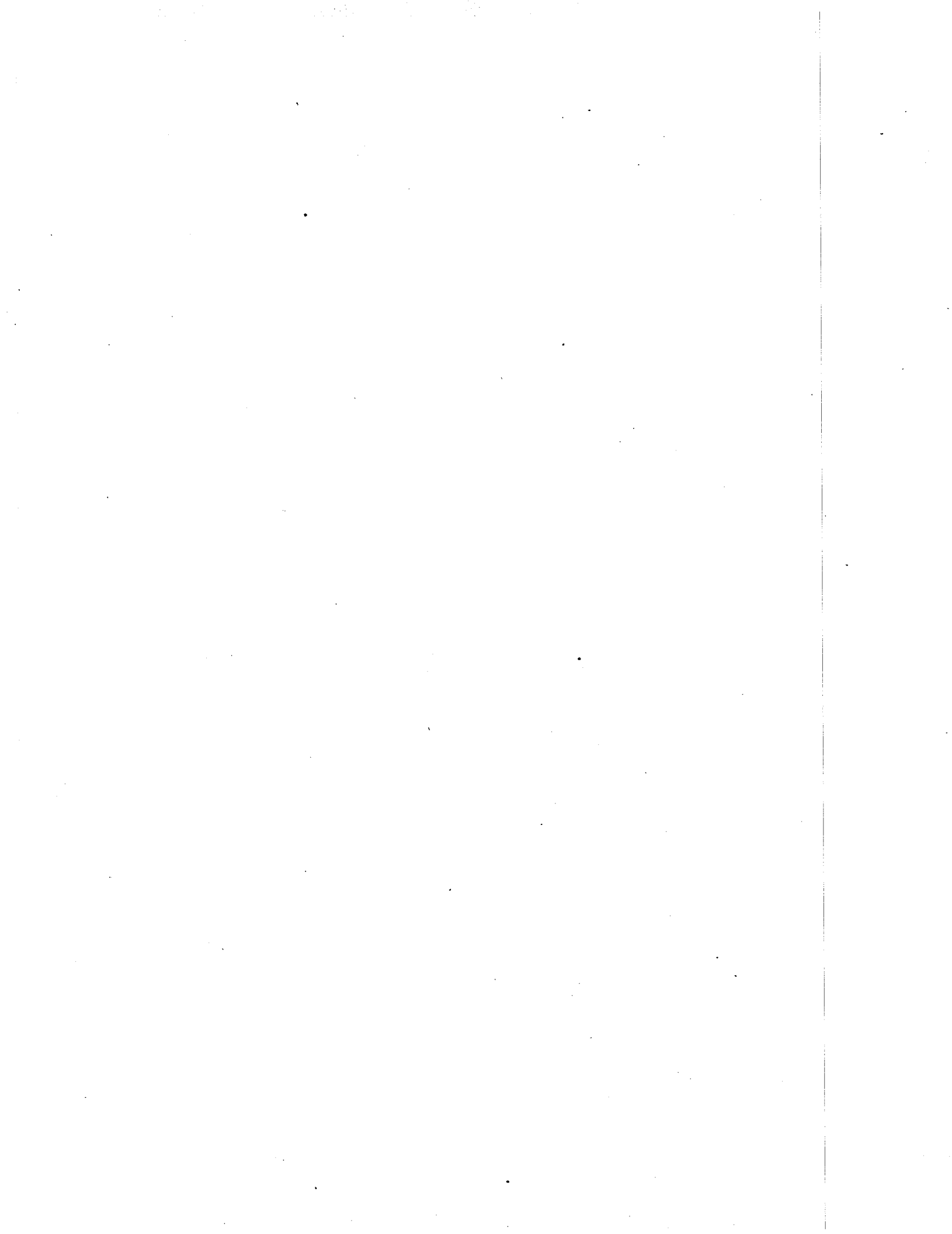
Command Cross-Reference Index

See also the more complete cross reference at the end of Chapter Four.

Add Mode	C-X M
Numeric Arguments	C-U
Backward, Beginning	C-A, C-B, C-P, M-A, M-B, M-V, M-[, M-<, C-X C-Z
Buffer	M-<, M->, C-X C-B, C-X B, C-X K
Capitalize Word	M-C
Center	C-L, M-S
Change	see Buffer, Delete, Insert, Replace, Windows
Character	, C-B, C-D, C-F
Copy Region	M-W see also Wipe, Yank
Delete	, C-D, C-K, C-W, see also Yank M-, M-C-K, M-D, M-K, M-\, C-X C-M, C-X K
Exit	C-X C-C
Files	C-X C-F, C-X C-R, C-X C-S, C-X C-W
Fill Paragraph	M-Q
Find	see Search, File
Forward	C-E, C-F, C-N, C-V, M->, M-E, M-F, C-X C-V
Go To	see Buffer, Move, Other Window

Indent	C-J see Center Line, Margins, Tabs
Insert	see beginning of Commands List, Quote
Kill	see Delete
Line	C-A, C-E, C-K, C-N, C-P, M-C-K, M-S, M-\
Lowercase Word	M-L
Margins	C-X ., C-X F
Mark	C-@, M-<SPACE>, M-H, C-X C-X
Modes	C-X C-M, C-X M
Move	see Backward, Copy, Forward, Kill, Yank
Next	see Forward
Other Window, Go.to	C-X O
Page	see Screen
Paragraph	M-H, M-Q, M-[, M-]
Previous	see Backward
Query	see Replace
Quit	C-X C-C
Quote	C-Q
Read File	C-X C-R
Redisplay	see Screen
Repeat	see Argument
Replace	M-C-R, M-R

Reverse	
see Search, Transpose	
Save File	C-X C-S
Screen	C-L, C-V, M-V, C-X C-V, C-X C-Z
Search	C-R, C-S
Sentence	M-A, M-E, M-K
Tabs	<TAB>, C-I, C-X <TAB>
Transpose	C-T, M-T
Undelete	
see Yank	
Uppercase Word	M-U
Windows	C-X C-V, C-X C-Z, C-X ^, C-X 1, C-X 2, C-X 0
Wipe	
see Delete	
Word	M-, M-B, M-C, M-D, M-F, M-L, M-U
Write File	C-X C-W
Yank	C-Y



2. Memory Management

A copy of the text that is being edited is stored in a buffer. The text appears to the user as a sequence of characters. All editing operations are specified relative to a place in the buffer. This place is called the point and it is always located between two characters (thinking this way eliminated the possibility of some fencepost errors). It is the responsibility of the memory management software to support buffers cleanly and efficiently.

It is assumed that the user will be presented with some sort of status display. This display will tell the user such things as the name of the buffer that he is editing, the name of the file that is being edited, and what modes the buffer is in (see section 4.5, page 37 for a discussion of what modes are). The interface to the memory management software includes operations to maintain this auxiliary information.

It is assumed that buffers are stored in the equivalent of main memory while the editing is being done. This means that the buffer is either in main memory (for very small machines), in the address space of the editor (for large address space virtual memory machines), or it can be mapped into the address space (for small address space virtual memory machines). Any of these cases will be assumed to be memory in this thesis. There are two commonly used techniques to manage memory in order to perform the editing efficiently. These techniques are known as buffer gap (store the text as an array of characters) and linked line (store the text as a linked list of lines) and will be discussed in following sections. Their discussion follows the more theoretical sections which cover the definition of the interface between the main editor and the memory management routines. Further discussion shows how the two schemes perform in a virtual memory environment and when multiple buffers are manipulated. Some closing remarks will be made about scratchpad memory and methods of reclaiming storage.

2.1 Data Structures

This section discusses the data needs of an editor. With two exceptions, all of the state of the editor is defined here. Thus, if this information is retained across invocations, you will have the ability to resume editing where you left off. Thus, the amount of work involved with editing can be reduced.

The other place where state information is kept is in the the screen manager. The screen manager to retains a knowledge of how buffers were displayed. Retaining this information allows the screen to reappear as the user left it. If the information is not retained, the screen manager will have to recalculate the display and this can be somewhat confusing. However, the editor will not lose any functionality if this state is left out.

The World contains the buffers in use by the editor. It is a circular list of BufferDescriptors and an indication of which buffer is the current one. In a PL/1-ish syntax:

```
declare 1 World,
        2 CurrentBuffer      pointer,
        2 BufferChain pointer;
```

Each buffer descriptor has several types of internal information.

```

declare 1 BufferDescriptor,
  2 NextChainEntry    pointer,
  2 BufferName         char(big) varying,

  2 Point             location,
  2 Length            fixed,
  2 Modifiedp        bit(1),

  2 FileName          char(big) varying,
  2 ModeName          char(big) varying,

  2 MarkList          pointer,
  2 Modelist          pointer,

  2 StorageData       pointer,
  2 ScreenData        pointer;

```

NextChainEntry is a mechanism for implementing the circular list of buffers. The list is circular because there is no preferred buffer and it should be possible to get to any buffer with equal ease. BufferName is a way for the user to be able to refer to the buffer.

Point is the current location where editing operations are taking place. It is of the data type location. The representation for this data type is implementation specific. For buffer gap editors, it is an integer, but for linked line editors, it is a (line pointer, offset) pair. Length indicates how long the buffer is in some reasonable unit (usually characters). Modifiedp is a flag which indicates whether the buffer has been modified since it was last written out or read in.

FileName is the name of the file system object which is currently associated with the contents of the buffer. ModeName is way to tell the user what modes are in effect. Typically, each mode will insert its name there as it is invoked. This information is not really implicit in the Modelist because there can be invisible modes (for example, autoloading commands) which use the mode mechanism for invocation but the user does not want to be made explicitly aware of them.

MarkList is simply a list of marks.

```

declare 1 Mark,
  2 NextMark    pointer,
  2 Name        {anything convenient, try small
                integers},
  2 WhereItIs   location;

```

Each mark has a pointer to the next one, a name, and a location within the buffer. Note that this list is not circular and it would probably help to keep it sorted by increasing location. The name is a way of distinguishing this mark from any other one associated with this buffer. This name is generated by the Create Mark routine and returned. It can thus be any convenient data type. Small integers will work quite well.

Modelist is a list of procedures to be invoked when this buffer is selected. See section 4.5, page 37 for a more complete discussion.

StorageData is a descriptor block which defines how the contents of the buffer are stored in

memory. The nature of this block is dependent upon the memory management algorithm used. ScreenData is a descriptor block which defines how the buffer appears on the user's screen. Its definition will become apparent in the discussion in the next chapter.

2.2 Marks

A mark is a named fixed point within a given buffer. A mark always points between the same two characters no matter what has been inserted or deleted around it. Marks are used for several different reasons.

- They remember a specific location for future reference. For example, a command might paginate a file. In this case, a mark would remember where the point was so that the command could return with the location of the point unchanged.
- They delimit a portion of text in conjunction with other marks or, more commonly, the point. This portion of text is called the region. This case would be used, for example, in a DeleteRegion command.
- They serve as bounds for iteration. Because they remain invariant when changes are made to the buffer, they can serve as a constant position to "head towards." An example could be the FillParagraph command. This command iterates through the buffer deleting and inserting whitespace (in the process, making each line as long as it can be without going past the right margin) until it reaches the end of the paragraph. A mark is used to remember where the end of the paragraph is. This usage is a variation on the region-delimiting usage, but it is worth noting in itself.

When an insertion is made at a mark there is a question about what to do with the mark (i.e. on which side of the inserted character it should end up). For the most part, the mark should move (i.e. be after the inserted character). However, there are good reasons for having it work the other way and so there are fixed marks, which remain before an inserted character.

An example of using fixed marks is to delimit an insertion. A routine could create both a mark and a fixed mark at the same location. Any inserted text would push the marks apart and end up between them. Thus, it is possible to keep track of what has been inserted.

2.3 Interface Procedures

This section defines the interface between the main part of the editor and the memory management routines. They will be described in terms of their logical function only, leaving out specific implementation details. An example of such a detail is a code variable which is returned and which indicates whether the operation succeeded. Also note that any data types mentioned (e.g. string) are intended to be canonical and no specific implementations are assumed. A <r> after a parameter means that it is returned by the procedure.

There is an important question as to exactly who allocates the data (the buffer descriptors and the buffers themselves). This issue is more language specific in the sense that certain languages specify an answer which must be used whether or not it is the right one. The procedures will be defined as if they own the data. If it is decided that they do not, it is relatively easy to include an extra argument

on each procedure call which identifies a descriptor of the object that the procedures are to manipulate.

```
InitWorld
SaveWorld(fileName)
LoadWorld(fileName)
```

InitWorld is the basic set-up-housekeeping call. It is called once, upon editor invocation. SaveWorld and LoadWorld implement the state-saving across edit sessions. SaveWorld is used to save the state of the editing session for later resumption. This operation might be quite expensive if it requires explicitly writing out all of the buffers to a large file or it might be very cheap in a virtual memory environment, where all that might be required is to set an external static variable to indicate that the environment is consistent. The possibility of multiple saved environments is interesting, but has not been implemented to my knowledge. It seems to be a nice way to work on several of tasks (not in the process management sense) at once.

If you are creating a "stripped down" editor then the save and load world routines will not do anything. They can be put in as stubs if there is a reasonable possibility that the editor will be embellished later.

```
CreateBuffer(BufferName)
DeleteBuffer(BufferName)
SetCurrentBuffer(BufferName)
SetCurrentBufferNext(BufferName <r>)
```

CreateBuffer is given a name and it returns after having created a buffer of that name. If the buffer already exists, it probably should signal an error of some sort to keep from bashing existing information. DeleteBuffer deletes a buffer. If the current buffer is deleted, the default buffer becomes the current one. (The editor is created with one default buffer called "Main" or something like that. It must always exist.)

Depending on the implementation language, we may be able to choose in the procedures that are being defined between including a buffer as an explicit parameter or having it implicit by setting an own variable that indicates which buffer is the current buffer. If buffers are changed often, it is worthwhile to include the buffer with each call. If, on the other hand, buffer switching is done infrequently, the overhead involved with setting a current buffer is more than acceptable. My experience has been that buffer switching occurs only rarely and so the extra call is worth it.

SetCurrentBuffer makes buffer BufferName the current one. SetCurrentBufferNext makes the next buffer in the circular list the current one and RETURNS its name in BufferName. This mechanism allows for iterating through all buffers looking for one which meets an arbitrary test.

Note that most of the above calls are really useful only if you have a multiple buffer implementation of the editor. In a single buffer editor, they are relatively useless and should be used only if there is a reasonable chance of expanding to a multiple buffer editor in the future.

```
SetModified(Flag)
GetModified(Flag <r>)
SetPointA(Location)
SetPointR(Count)
```



```
GetPoint(Location <r>)
GetLength(Size <r>)
```

These routines deal with several variables. They allow setting and asking for the point, the current buffer length and the state of the modified flag.

The modified flag provides an indication of whether the buffer has changed. It is set implicitly by any buffer change operation (principally insertion and deletion) and cleared automatically by writing the contents of the buffer to a file. The procedures to set or clear it explicitly are provided as this ability will ordinarily be used by the redisplay code (see section 3.6.2, page 27 for the discussion of what it is used for).

Note that there are two flavors of the SetPoint routine, designated "A" and "R". They both do the same logical operation, but the "A" version interprets its argument as an absolute position within the buffer and the "R" version interprets its argument as an offset relative to the current position of the point. (Negative values indicate a backward offset.) Due to the definition of the location type, the "A" version does not take an integer value as the location and so one usage is not readily simulatable in terms of the other.

```
Insert(String)
Delete(Count)
GetStringA(Location,Length,String <r>)
GetStringR(Count,Length,String <r>)
```

These routines manipulate and examine the buffer. Insert inserts a string into the buffer at the point. The point is left at the end of the inserted string. Delete removes abs(Count) characters from the buffer. (Negative counts delete before the point).

GetString returns the string starting at the specified location and Length characters long. There are both absolute and relative versions of this routine.

```
Search(String,Location <r>,Flag <r>)      F | B   A | R
FindFirstIn(String,Location <r>,Flag <r>) F | B   A | R
FindFirstNotIn(String,Location <r>,Flag <r>) F | B   A | R
LookingAtP(String,Location,Flag <r>)      A | R
```

There are a total of fourteen routines in this section, but they have been listed in an abbreviated form for convenience. These are search routines and each form can head either forward (F) or backward (B) and with the returned location either absolute (A) or relative (R) to the point.

Note that these routines are not necessary as their action can be readily simulated by using the other defined routines. However, they have been included in the discussion because they are useful and because they are often implemented in the same level as the other memory management routines for having lower level access to the buffer will speed up their execution.

Search looks for the first occurrence of the string in the buffer, starting at the current point and heading in the specified direction. It returns a flag saying whether the string was found and an indication of the string's location. The location returned is the location of the end of the found string (either absolute or relative) in the direction of the search. (For a backward search, the location is the beginning of the actual string.) This definition implies that repeated searches will stop at successive

instances of the string.

FindFirstIn searches for the first occurrence of any of the characters in the string. For example, FindFirstIn("0123456789",...) would return the location of the first digit encountered. Like Search, it returns a flag as to whether a match was made and the location of the match. Unlike Search, the location returned is at the beginning of the match and not the end. Successive applications will thus return the same position. The difference in behavior between Search and FindFirstIn is a function of their different uses. FindFirstIn is used to parse through text slowly, and zero and one character strings must be handled properly. These definitions facilitate that handling. FindFirstNotIn works in exactly the same manner as FindFirstIn except that it matches on any character not in the string. For example, the following code fragment implements a forward word operation.

```

alphabet="abcdefghijklmnopqrstuvwxy";
FindFirstInFA(alphabet, location, flag);
SetPointA(location);
FindFirstNotInFA(alphabet, location, flag);

```

The first Find operation will skip over any non-word characters to the beginning of a word. The next one will skip to the end of the word. (Note that the alphabet variable should also have the digits, uppercase, and several special characters in order to work as one would intuitively expect. Note also that the selection of special characters will in general be language-specific. Further, no checks were made for string-not-found, etc. Thus, it is not an example of finished code.)

An alternative way of defining these operations is to have them automatically set the point instead of returning a location. (The flag must still be returned.) If the string was not found, the point would not be moved. The choice of a method of implementing these routines is a matter of taste.

LookingAtP has a much more simple definition. It compares String against the sequence of characters in the buffer starting at Location, returning the true/false answer in Flag.

```

GetHpos(Column <r>)
SetHpos(Column)

```

(These routines, like the previous set, are not necessary but useful.) GetHpos returns the column that the point is in, after taking into account tab stops, etc. It does not take into account the screen width as it should not make any difference to the editor how big the terminal is. SetHpos moves the point to the desired column, stopping at the end of a line if it is not long enough. If there is no character at the desired column (due to tab stops), it uses the next higher available column position.

```

SetFileName(FileName)
GetFileName(FileName <r>)
WriteBuffer
ReadBuffer

```

These routines interface between the buffer and the file system. The FileName routines set and return the file object (in general a string--the file name). At the user interface, the editor might implement an intelligent "default and guess" interpretation of the file name so as to make life easier for the user, but doing so does not affect this level of code. This general area is one where system-specific conventions become significant.

WriteBuffer writes the contents of the buffer out to the file name associated with the buffer. Any conversions between internal format and what the file system requires will be done at this time. Also, the buffer modified flag (ModifiedP) will be cleared.

ReadBuffer reads the file into the current buffer. There are two choices about how to do the read operation. Both directions will be discussed along with their ramifications for the other parts of the editor. They are not both implemented because it is desirable to keep the number of primitives to a minimum.

First, it can replace the contents of the buffer with the contents of the file. If it does so, the buffer modified flag will be cleared automatically. The editor will want to check on what it is replacing. If the previous contents of the buffer have been modified, the user should be asked what to do (e.g., whether he is making a mistake).

Second, it can insert the contents of the file into the contents of the buffer at the point. In this case, the first method can be simulated by explicitly deleting all of the buffer and then reading. The buffer modified flag will have to be manually cleared. The same policy of asking the user what to do with modified buffers should be followed. The advantage behind this method is that it allows the easy implementation of the insert file command. The first method requires the allocation of additional space and then the copying of the data; a luxury that may not be available on smaller systems. This second method is thus preferred.

```

CreateMark(MarkID)
CreateFixedMark(MarkID)
DeleteMark(MarkID)
SetMark(MarkID,Location)
GetMark(MarkID,Location <r>)
CompareLocation(Location1,Location2,Result <r>)

```

These routines manage marks. They allow for creating both ordinary and fixed marks, deleting marks, and setting and evaluating them. Note that except for creating them, there is no difference in usage with these routines between ordinary and fixed marks (although their behavior will, of course, differ).

SetMark merely sets the location of the mark to Location. (A relative version of this routine can be supplied, if desired.) GetMark returns the current location of the mark. It should be used directly and not assigned into a variable as its value can change across some buffer operations. These operations are Insert, Delete and ReadBuffer.

CompareLocation allows the comparison of any two marks or the point and the mark to be done without being aware of the specific scheme chosen. It takes two Locations as arguments and returns the sign (+1, 0, -1) of the result of Location1 - Location2.

```

SetModeName(ModeName)
GetModeName(ModeName <r>)
AppendModeList(Procedure)
DeleteModeList(Procedure)
InvokeModeList

```

These routines manage the multiple mode capability. The ModeName is a string which can be

displayed to remind the user what is going on. It does not affect anything else.

Append, delete, and invoke operations are all supplied. It is generally bad form to define the modes so that it matters in which order the procedures are called, but there do arise such occasions. Therefore, the procedures should be called in the order that they are appended onto the list. Checks should be made to insure that a procedure is not put on the list more than once. Again, see section 4.5, page 37 for a complete discussion of modes.

2.4 Buffer Gap

This section discusses the implementation of one of the two ways of implementing the memory management functions.

A buffer gap system stores the text as two contiguous sequences of characters with a (possibly null) gap between them. It thus uses memory efficiently as the gap can be kept small and so a very high percentage of memory can be devoted to actually storing text. Changes are made to the buffer by first moving the gap to the location to be changed and then inserting or deleting characters by changing pointers.

In more detail, here is an example buffer which contains the word "Massachusetts".

0	1	2	3	4	5	6	7	8		9	10	11	12	13																		
	M		a		s		s		a		c		h		u						s		e		t		t		s			

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16																
30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45																	
															P						GS							GE				

There is a lot of information here which needs explaining. First, the buffer is 13 characters long and it contains no spaces. The blanks between the "u" and the "s" show where the gap is and do not indicate that the memory has spaces stored in it. The point is between the "a" and the "c" at location 5 and is labeled with a "P" in the bottom line (legal values for the point are the numbers from zero to the length of the buffer). There are also three different sets of numbers (coordinate systems) for referring to the contents of the buffer.

First is the user coordinate system. It is displayed above the buffer. The values for it run from 0 to the length of the buffer. As you will note, the gap is "invisible" in this system. The coordinates label the positions between the characters and not the characters themselves. Thought of in this way, the arithmetic is easy. Thought of as labeling the characters, the arithmetic becomes fraught with special cases and ripe for fencepost errors.

Second is the gap coordinate system. It is displayed immediately under the dashed line. The values for it run from 0 to the amount of storage that is available and it, too, labels the positions between the characters (or rather, storage cells). The internal arithmetic of the buffer manager is done in this coordinate system. The start of the gap (labeled "GS" in the bottom line) is at position 8 and the end of the gap (labeled "GE") is at position 11.

Conversion from the user coordinate system to the gap coordinate system is quite easy. If the location (in the user coordinate system) is before the start of the gap, the values are the same. If the

location is after the start of the gap (NOT the end of the gap!), the location in the gap coordinate system is $(\text{GapEnd} - \text{GapStart}) +$ the location in the user coordinate system. It is a good idea to isolate this calculation either in a macro or a subroutine in order to enhance readability. Most routines (e.g. Search) will then use the user coordinate system even though they are essentially internal.

The third coordinate system is the storage coordinate system. It is the bottom row of numbers in the diagram. It is the means whereby the underlying storage cells are referenced. It is labeled from X to X + the amount of storage that is available. The origin (the value of X) 30 was chosen to be 30 here to help distinguish between the various coordinate systems. Its absolute value makes no difference. Note that it labels the cells themselves and so caution must be taken to avoid fencepost errors.

A buffer gap system has a very low overhead for examining the buffer. The reference (GetChar) comes in in the user coordinate system and the location is converted to the gap coordinate system. The cell is looked up and the contents returned. Essentially, one compare and a few additions are required. The purpose of the conversions is to make the gap invisible. Note that in no case is any motion of the buffer necessary.

There is more of an overhead associated with inserting or deleting a character. In this case, the gap must be moved so as to be at the point. There are three cases:

1. The gap is at the point already. No motion is necessary.
2. The gap is before the point. The gap must be moved to the point. The characters after the gap but before the point must be moved. Thus, $\text{ConvertUserToGap}(\text{Point}) - \text{GapEnd}$ characters must be moved. This quantity is numerically $\text{point} - \text{GapStart}$.
3. The gap is after the point. The gap must be moved to the point. The characters after the point but before the gap must be moved. Thus, $\text{GapStart} - \text{ConvertUserToGap}(\text{Point})$ characters must be moved. This quantity is numerically $\text{GapStart} - \text{point}$.

After the gap has been moved to the point, insertions or deletions can be effected by moving the GapStart pointer (or the GapEnd pointer--it makes no difference). A deletion is a decrementing of the GapStart pointer. An insertion is an incrementing of the GapStart pointer followed by placing the inserted character in the storage cell.

Note that after the first insertion or deletion, further such operations can take place with no motion of the gap (it is already in the right place). Further, the point can be moved away and back again with no motion of the gap taking place. Thus, the gap is only moved when an insertion or deletion is about to take place and the last modification was at a different buffer location.

This scheme has a penalty associated with it. The gap does not move very often, but potentially very large amounts of text may have to be shuffled. If a modification is made at the end of a buffer and then one is made at the beginning, the entire contents of the buffer must be moved. (Note, on the other hand, that if a modification is made at the end of a buffer, the beginning is examined, and another modification is made at the end, no motion takes place.) The key question that must be asked when considering this scheme is, when a modification is about to be made, how far has the

point moved since the last modification?

Sidnote Calculation. How far can the point be moved before the shuffling delay becomes noticeable? Assume 1/10 sec. is noticeable and that it is a dedicated system. Assume 1usec, 8 bit wide memory. Assume 10 memory cycles per byte moved (load, store, eight overhead cycles for instructions). Then, 10,000 bytes can be moved with a just noticeable delay.

Because of the locality principle, it seems reasonable to conclude that for almost any rational buffer size the average distance moved will be less than 10K bytes and so the shuffling delay will not be noticeable.

2.4.1 Gap Size

Note that the size of the gap does not affect how long the shuffling will take and so it should be as large as it can be. Typically, it is all of the otherwise unused memory. In that case, when the gap size goes to zero, there is no more room to store text and the buffer is full.

2.4.2 Multiple Gaps and Why They Don't Work

Assume that we were still uncomfortable with the shuffling delay and a possible fix was put forth. This fix would be to have, say, ten different gaps spread throughout the buffer. What would the effects be? The idea behind this discussion is to help in understanding the buffer gap system by seeing how it fails.

First, the conversion from the user to the gap coordinate system would be more complicated and take longer. Thus, some ground has been lost. However, this is a small loss on every reference in order to smooth out some large bumps, so it might still be a reasonable thing to do.

Second, the average amount of shuffling will go down, but not by anywhere near a factor of ten. Because of the locality principle, a high percentage of the shuffling is of only a short distance and so cutting out the "long shots" will not have a large effect.

Third, unless the writer is very careful, the gaps will tend to lump together into a fewer number of "larger" gaps. In other words, two or more gaps will meet with the GapEnd pointer for one matching the GapStart pointer for another. There is just as much overhead in referencing them, but the average amount of shuffling will increase.

On the whole, the extra complexity does not seem to return proportional benefits and so this scheme is not used.

2.4.3 The Hidden Second Gap

On two-dimensional memory systems such as Multics, a second gap at the end of the buffer is provided with almost no extra overhead. The key to this gain is that the buffer is not stored in a fixed-size place. Rather, the size of the memory that is holding the buffer can also increase.

The extra overhead is a check to see whether a modification is taking place at the end of the buffer. If so, the modification is made directly with the EndOfAvailableStorage (the buffer runs

from X to $X + \text{EndOfAvailableStorage}$) variable serving to note that the change has taken place.

This change has more of an effect that might at first be apparent because a disproportionately high percentage of modifications take place at the end of the buffer. This distortion is due to the fact that most documents, programs, etc. are written from beginning to end and so the new text is inserted at the end of the buffer.

The overhead for this change is low because the check for the end of the buffer was already there. There is no problem of the gaps coalescing because one of them is pegged into place. The gains are not all that great, but neither are the costs and so it is used. This technique is also usable with some implementations of multiple buffers.

2.5 Linked Line

The other method of memory management that we will discuss is called linked line. It stores the buffer as a doubly linked list of lines. This method is especially useful with languages such as Lisp which provide memory management facilities integral with the language.

Each line in the linked list has several pieces of information in its header. Not all of these pieces are required, but they can help greatly in managing the buffer. The pieces of information are:

```

NextLine      pointer      /*32 bits*/
PreviousLine  pointer      /*32 bits*/
Length        fixed        /*16 bits*/
/*presumably no SINGLE line will be >64K characters*/
Line          char         /*the line itself*/

```

optional fields:

```

AllocatedLength fixed      /*16 bits*/
Version         fixed      /*32 bits*/
Marks           pointer    /*32 bits*/
TextPtr         pointer    /*32 bits*/

```

The NextLine and PreviousLine fields implement the doubly linked list. The length field is, clearly, the number of characters in the line. These, along with the line itself, are all that are required in order to implement the linked line scheme. The other fields are a help in making the scheme efficient and some of them are very valuable to include.

The AllocatedLength field indicates how much memory is allocated to storing the line itself. Thus, an allocate/free combination are not required each time a character is inserted or deleted. For example, a memory allocation block size of 16 bytes has been used in some implementations of this scheme. AllocatedLength will then be either 0, 16, 32, 48, 64, etc. The allocate/free combination is only required every time the line crosses a 16 byte boundary, a considerable savings in overhead.

Allocating memory in 16 byte chunks cuts down significantly on fragmentation. It will almost certainly be possible to run without a compactifying garbage collector. See the discussion of scratchpad memory (section 2.9, page 17) for further information.

The version field is for use by the redisplay code and is an optimization to make it run faster. It

will be discussed with the rest of the redisplay process. It serves the purpose of specifying a unique id for the line.

Using integer-valued buffer positions is hard with the linked line scheme. Instead, a (line pointer, offset) pair are used. Marks are then always associated with a line and can thus be merely strung in a list associated with the line that they are on. With this implementation, less time is required to update the marks because only those that are on the line can possibly be changed. Note that there should still be a central listing of all marks in order to facilitate finding any given one and that mark ids should be unique within a buffer.

Finally, instead of storing the text of the line with the header, it can be separately allocated. The TextPtr field is then used to remember where the text is. This ability is especially useful when several places point to the header and properly updating them whenever the line is reallocated is difficult.

In summary, the most useful fields are NextLine, PreviousLine, length, AllocatedLength, and either version or the mark list. These fields can fit within one 16 byte allocation block.

The operation of a linked line scheme is quite straightforward. New lines, when created, are simply spliced into the list at the appropriate place. (Note that no characters are stored to indicate line breaks). If the new line is in the middle of an existing line, some movement of the text on the end of the old line to the new line is all that is required.

The line itself is stored as a packed array of characters. Inserting or deleting text is done by scrolling the line after the point of modification. Clearly, this scheme is very inefficient with large line lengths.

The reason why the length fields were 16 bits long is not obvious. After all, only rarely will a document have even 256 character lines. But people occasionally edit rather strange things, including object files. One cannot rely on encountering new line characters at reasonable intervals in such files. Thus, the extra size.

2.5.1 Storage Comparison

Storage requirements for a linked line scheme are somewhat higher than for buffer gap. A buffer gap scheme requires one or two new line characters per line, and a small amount of fixed storage (GapStart, GapEnd, etc.).

Linked line requires, in a reasonable implementation, one 16 byte block plus an average of 8 bytes lost due to fragmentation for each line. On the other hand, large amounts of text will never have to be moved.

2.5.2 Error Recovery Comparison

Recovering from errors (an unexpected program termination, for example) is relatively easy and fail soft in a buffer gap. In general, the start and end of the buffer are findable if a marker is left around the buffer (say, a string of sixteen strange (value 255) bytes) and the buffer is everything between them. The gap can be recovered and manually deleted by the user or, if it, too, is filled with a special marker, it can be automatically deleted.

Linked line management is harder to recover. Recovery is greatly aided by erasing freed memory. Basically, you pick a block at random and examine it. If it can be parsed into a header (i.e., the pointer values, etc., are reasonable), continue (a careful selection of header formats will help). Otherwise, pick a different block. You can then follow the next and previous pointers and parse them. If this works three or four times in a row, you can be confident that you have a handle on the contents. If a header doesn't parse, it is because it is either a part of a line (either pick again at random or go back one chunk and try again) or a header that was being modified (in which case you are blocked from continuing down that end of the chain). In the latter case, go in the other direction as far as possible. You now have one half of the buffer. Repeat the random guess, but don't pick from memory you already know about. You should get the other half of the buffer. Leave it to the user to put them together again. If the freed blocks are not erased, the chance of finding a valid-looking header that points to erroneous data is very high.

2.6 Multiple Buffers

How do buffer gap and linked line schemes implement multiple buffers? There is a variety of choices:

```

intertwining (linked line only)
separate storage for each buffer
    large address space (therefore paged)
        structured
        non structured
    small address space
    special cases

```

Intertwining is an option that is only open to linked line. In this case, all allocation is done out of a common pool and so, over time, the buffers tend to "intertwine" (i.e., the lines of one buffer are mixed in with the lines from other buffers in physical memory). Such an approach tends to maximize the density of text and thus make the most efficient use of memory. It also assumes that a large address space is available. (See also the discussion in the next section about paged environments.)

Separate buffer space means that each buffer is allocated out of its own area and that all of a buffer's area is contiguous. Thus, the address space is cut up into separate sections for each buffer.

If a large address space is available, the cutting up can be done one of two ways. If the address space is structured (as in Multics), the operating system takes care of managing such things automatically. If the address space is not structured (as in Vax/VMS), the memory management scheme can reserve fixed regions of the address space for separate buffers, each more than large enough for any reasonable file.

If the address space available is too small to reserve effectively, the memory management scheme will have to keep track of all of the buffers and map them into and out of the available address space as needed. Caution must be taken to avoid requiring that only one buffer be in the address space, as a multiple window editor must be able to scan multiple buffers. In addition, auxiliary buffers will be needed from time to time (e.g., for copying text from one to another).

Managing multiple buffers is relatively easy. They are treated as a set of buffers, only one of which can be accessed at a time. See the earlier section on buffer data structures (section 2.1, page 3).

2.7 Paged Virtual Memory

How well do the two schemes perform in a paged virtual memory environment?

The buffer gap scheme works very well in general. Its highly compact format allows for accessing large parts of the buffer with only a few pages in memory. Its sequential organization also implies that it has a very good locality of reference and so the nearby pages are heavily referenced and likely to be around.

Its major problem is the large amount of shuffling that must be done in some cases. A move of the whole buffer implies that the whole buffer must be swapped in and--most likely--swapped out again. (A search of the whole buffer also requires this swapping, but the user asked for it and no management scheme can search linearly through memory that is on disk. Therefore, the user should expect lesser response.) If the memory manager is built into the operating system, some interesting hackery can be done with the page table to "move" all of a chunk of memory by one or more pages by moving page table entries. The existence of this example implies that such a function might well be desirable to include in a future set of operating system calls ("insert n pages after page x and scroll through page y"--delete n pages is implicit in this and it only affects part of the address space).

In a tight memory situation, the buffer gap scheme does as well as can be expected. The nearby sections of the buffer will be around because of locality of reference, but anything far away can take a while to get to.

A linked line scheme does not perform as well overall. First, if an intertwining multiple buffer scheme is used, one may as well forget performance in a tight memory situation. The intertwining can use different parts of each page for storing different buffers. Thus, when considering any given buffer, the page size is effectively reduced.

Even in a separated buffer scheme, the data is not as tightly packed overall (the headers and fragmentation) and so some performance is lost. Also, the linked lines can be anywhere in a large portion of memory and so the density of nearby lines can range from good to very low. Finally, even if a desired line happens to be on an in-memory page, in order to get there (via the links), you will probably have to swap in several additional pages and, in the process, may even swap the desired page out!

The primary advantage that linked line has is that it never requires moving large sections of the buffer. Thus, if memory is not tight, the entire buffer can fit in memory and performance will be very good.

2.8 Editing Extremely Large Files

Extremely large files come in two flavors. First are files that are so large that reasonable assumptions break down. Such things tend to start happening about 64M bytes or so. At that point, even simple things (e.g. string search) tend to take several minutes to run on a fast processor with the whole file in memory.

Although there are one or two interesting hacks to stay alive, life is simply not bearable when trying to edit such a large unstructured file. The alternative (which large data base people have known about for years) is to structure the file. This alternative is not that unpalatable because an

unstructured editor can still be used to edit the subpieces of structure. The other reason why this is not that much of a problem is that there aren't all that many gigantic files to edit. The vast majority of files are much smaller. Gigantic files call for special tools for manipulating them.

The other flavor is more applicable to microprocessors where an extremely large file might be 100K bytes. The reason why it is considered so large is that the disk to store it on might be only 50K bytes, or there might only be magnetic tape for permanent storage. Thus, a 100K file would tax the hardware resources severely.

The basic way of dealing with such files is to break them up into chunks and edit the chunks separately (the TECO yank command is an example of this). In general, you can only proceed forward through the file in any given edit session because of the problems involved with the file size changing as the edit progresses. Either a marker byte (\uparrow L is commonly used) or a character count (not as polite to the user) can be used to determine where the file breaks are to occur. This method requires an input and output file to both be available and open at the same time. A crash preserves the input file and some of the output file. Thus, editing a 100K file requires up to 200K of storage. This is the only method that works on magnetic tape.

The next method allows full access to the file without breaking it up in any way. It requires three files (input, output, and backup) to be open simultaneously. As you proceed through the file, it edits from the input to the output file. However, when you reverse direction, it reads from the output file onto the third, backup file (it does not modify the input file, thus ensuring its integrity in the case of a system crash). Note that the data is stored in the backup file in reverse order! Preferably, file i/o is done in blocks and only the order of the blocks needs to be reversed, not the contents of the blocks themselves. When you switch to going forward once again, the backup file is read until it is exhausted and then use of the input file is resumed. This method allows for simulating a very large buffer as the file management can be done invisibly. Thus, the user can edit a 100K file with much less physical memory. Note that the swapping can be slow!

The final method that is available is to simulate demand paging by breaking a buffer gap scheme up so that there are many small buffers. Each buffer is then paged to disk independantly. If a buffer should fill up, it can be split up into two buffers and insertions can continue. No large motion of text is ever required, but memory is lost.

In none of these systems is linked line acceptable. Memory is assumed to be very tight and the overhead of the extra headers is not acceptable.

2.9 Scratchpad Memory

Scratchpad memory contains the temporary variables allocated by the editor. Because of the transient nature of these variables, it is allocated and freed often. It is used to hold the buffer descriptors, string variables, and--in the linked line scheme--the buffers themselves. The scratchpad memory management required for text editing is relatively simple, but there are some general considerations that are worth mentioning. There aren't too many buffer descriptors and they are of a well known size so they are easy to manage. The string variables can range from being null to being entire buffers. Thus, they can cause fragmentation quite easily. The linked line formats have already been discussed.

In a large address space system, two buffers worth of address space should be devoted to scratchpad storage (to allow for putting an entire buffer there, which takes one buffer worth, and because space is allocated in integer buffers worth). In a small address space system, large operations are typically done character at a time because memory itself is usually at a premium. Therefore, the amount of scratchpad storage needed can be quite small. In any system where the editor can be dynamically extended (see the Command Loop chapter), scratchpad storage needs can vary dramatically and are not generally predictable in advance.

Allocating memory in chunks helps prevent fragmentation, therefore not usually requiring a compactifying garbage collector. If memory becomes badly fragmented, a compaction is required. In a linked line scheme, compaction eliminates the possibility of using the line pointers as unique ids (they change). Such unique ids are used by the redisplay algorithm.

3. Incremental Redisplay

The most visible part of a screen-oriented text editor is the redisplay process. This is the section of code that keeps the current contents of the buffer accurately displayed on the user's terminal. It has the additional goal of performing this function in such a way that a minimum or near minimum amount of clock time is required in order to fulfill this purpose. Clock time is a combination of transmission time, cpu time, and disk access time which is perceived by the user as the delay from when he enters the command to when the redisplay is finished.

In general, the contents of the buffer will change only a small amount during the basic read command - evaluate it - do redisplay loop. The screen will then only have to be changed by a small amount in order to reflect the changed buffer contents. Hence, the algorithms concentrate on incrementally redisplaying the buffer and the entire process is referred to as incremental redisplay. Fortunately, it turns out that in cases where the buffer is changed drastically, the increment-oriented approach to redisplay works quite well and so there is no need for multiple algorithms.

Our discussion of the incremental redisplay process assumes a model of the system where the editing is done on a main processor which communicates with a terminal. If the main processor is the same as the terminal, the bandwidth of the communication channel can be thought of as being very high. The incremental redisplay process is an optimization between cpu time and I/O channel time, with a few memory considerations thrown in. The primary constraint is the speed of the I/O channel. Typical speeds that are currently available are 30 characters/second, 120 cps, and 960 cps. There are also memory mapped terminals which run at essentially bus speeds. Equivalent speeds can be derived and run in the 100 to 50,000 cps range.

A typical video terminal has a 24 x 80 character screen. At 30 cps, it will thus take three seconds to print a line and over a minute to refresh the whole screen. At 120 cps, less than one second is required to print a line and about twenty to refresh the screen. At 960 cps, it will take only one or two seconds to refresh the screen. The speed of the communication greatly affects the amount of optimization that is desired. At 30 cps, even one extra transmitted character is painful to the user, while at 960 cps reprinting entire lines does not take an appreciable amount of time. One dimension of the optimization is thus clear: the importance of optimizing the number of characters sent increases in proportion to the slowness of the communication line.

A user interface issue arises at this point. While it is acceptable from a clock time point of view to reprint entire lines, users do not like to see text which has not changed in the buffer "change" by being reprinted. The flickering that is generated by the reprinting process attracts the user's attention to that text, which is undesirable (the text has not, after all, changed). Thus, avoiding extraneous flickering and movement of text is good. Even with infinitely fast communications and computation, incremental redisplay will still be a desirable feature.

Cpu time must be spent in order to perform these optimizations. If the cpu time that is spent exceeds some small amount of clock time, response will be annoyingly sluggish (and that is not good). It is therefore desirable to minimize the cpu time that is spent on optimizing the redisplay. At this point, the speed of the communication line makes a difference. If the line is slow, extra cpu time can and should be spent (at 30 cps, it is worthwhile to spend up to 30 msec. of cpu time to eliminate one character from being transmitted (which takes about 30 msec.)). However, at higher speeds it is generally not practical to heavily optimize as it can easily take longer to compute the optimizations

than to transmit the extra text. This relaxation of the optimization is subject to the user interface constraint outlined above. Memory size constrains the optimization as well. One technique used is storing the entire screen, character by character. This technique works quite well; however, where memory is tight this technique will prove too expensive to implement.

3.1 Line Wrap

There are some more pragmatic considerations involved in the design of the redisplay process. The first of these is line wrap.

Although the editor is editing a one-dimensional stream of text, this text must be placed on a two-dimensional screen in such a way that the user can understand it. There should be no constraints made by the redisplay process on the length of lines. Additionally, there are no commands to "position the screen" or anything of the sort. IT IS THE RESPONSIBILITY OF THE REDISPLAY PROCESS TO HAVE THE SCREEN SHOW MEANINGFUL INFORMATION AT ALL TIMES. The user has almost no control over this function at all, and should not need to. If commands have to be entered in order to obtain feedback, those are commands that are not doing productive editing.

There are two different ways to handle very long lines. One way is to have these lines be clipped at the right hand edge of the screen and then have some indication that the clipping is occurring. The other is to wrap the lines to the next line (i.e., the text that does not fit on one screen line is placed on the next). The first method is acceptable, but not very well human-engineered. Typing text in the middle of a line causes the line to spill and visually lose characters. This losing of characters causes uncertainty in the user's mind about what exactly is happening. In addition, it is never possible to see a long line in its entirety.

The second method is slightly less "clean" when displayed on a screen as wrapped lines will be around, but it does not suffer from either of the above problems. Inserting text might cause a line to wrap (an annoying process) but no text vanishes. Also, long lines are always visible. Finally, wrapped lines are usually only a temporary phenomenon, because most people prefer line widths in the 65-80 character range and this range fits on most terminals. Thus, the wrapped lines appear mainly during editing and will normally go away. Note that it is during the editing process that users most need the feedback. Thus, the line-wrapping method seems to be the best one to use.

In any method, care must be taken to make sure that the pathological case of very long lines works properly. Although rare, non-text (e.g., object code) files are sometimes examined with the editor. These files generally do not break up into reasonable-sized screen lines (a newline indicator might not occur for two or three thousand characters in an object file). Thus, a single line of text might more than fill up the screen. Provisions must be made in the redisplay code to allow the screen to nonetheless be positioned into the middle of such a line.

3.2 Multiple Windows

It is useful to be able to see more than one buffer (or different parts of the same buffer) simultaneously. For example, you can then examine documentation while writing a procedure call. In general, it is not too difficult to set up the redisplay to perform this multiple windowing. The few necessary details will be mentioned in the discussion of the algorithms themselves. Care must be

taken that modifications made while in one window are reflected in any other appropriate windows.

3.3 Terminal Types

The redisplay process is the way to communicate to the user. It also has a strong interest in taking advantage of whatever features are supplied by the terminal in order to reduce the time taken for a redisplay. This section will undertake a brief discussion of the various classes of terminals available and how various features affect the redisplay process.

3.3.1 TTY and Glass TTY

A TTY is a canonical printing terminal. Printing terminals have the property that what is once written can never be unwritten. A glass TTY is the same as a TTY except that it uses a screen instead of paper. It has no random cursor positioning. Incremental redisplay for such a terminal usually maintains a VERY small window (e.g., one line) on the buffer and either echos only newly typed text or else consistently redisplay that small window. Once a user is familiar with a display editor, however, it is possible--in a crunch--to use it from a terminal of this type. This is not generally a pleasant way to work.

3.3.2 Basic

A basic terminal has, as a bare minimum, some sort of cursor positioning. It will generally also have clear to end of line (put blanks on the screen from the cursor to the end of the line that it is on) and clear to end of screen (ditto, but to the end of the screen) functions. These functions can be simulated, if necessary, by sending spaces and newlines. A typical basic terminal is the DEC VT52.

Such terminals are quite usable at higher speeds (960 cps) but usability deteriorates rapidly as the speed decreases. It requires patience to use them at 120 cps and a dedication bordering on insanity to use them at 30 cps. Terminals which do not have clear to end of line are even worse.

3.3.3 Advanced

Advanced terminals have all of the features of basic terminals along with editing features such as insert/delete line and/or character. These features can significantly reduce communication time for common operations. Typical terminals in this category are the HDS Concept 100, the Teleray 1061, and the DEC VT100.

These terminals are, of course, quite usable at 960 cps and similar speeds. Due to the reduced need for communication line bandwidth, at lower speeds they are more usable for editing than anything else. At 120 cps, editing text is relatively painless, but merely examining text takes place at a quite slow speed. At 30 cps, even editing is barely acceptable.

There is a subtle difference among some of the advanced terminals. The VT100 supports a scroll window (move lines x through y up/down n lines) feature while the 1061 supports insert/delete lines. Scroll window is more pleasing to see when there is some stationary text being displayed at the bottom of the screen. With insert/delete line, the appropriate number of lines must be deleted and then inserted; the text at the bottom thus jumps. Scroll window does the whole thing as one operation and does not cause the bottom to jump.

The C100 has an interesting feature. It is a fully windowed terminal and thus all operations can be confined to only affect a designated area on the screen. Insert/delete line operations thus do not cause the bottom text to jump and it is even possible to have two windows side by side as the clear to end of line operation does not affect the text in the adjoining window. The window management software thus has much more flexibility in what can be done while remaining within reasonable transmission time constraints.

3.3.4 Memory Mapped

This section covers a wide range of terminals. Their common characteristic is that the entire screen can be read or written at near bus speeds. Typically, this means that the terminal is "built in" to the computer that is running the text editor. In addition, this computer is often a dedicated one, running only one user's processes. Examples of this type of terminal are the Knight TVs (at the MIT AI lab), the Lisp Machine displays, and the wide variety of memory mapped displays available for microprocessors.

The use of memory mapped terminals has several implications for the redisplay process. First, many of the advanced features are typically not available. However, the terminal I/O is so fast that they can be emulated very quickly. Second, it is possible in some cases to use the screen memory as the only copy of the screen. Thus, if reading from the screen does not cause flicker (but writing does), the screen can be read and the incremental redisplay process will run and compare the buffer against it, changing it only when necessary. Finally, if you can write to the screen without flicker, the redisplay process merely boils down to copying the buffer into the screen as doing so is always faster than comparing. Any memory mapped terminal which has a slow access time should be thought of as a basic terminal for the purpose of redisplay algorithms.

3.3.5 Terminal Independent Output

A full discussion of this topic is beyond the scope of this thesis. [Linhart] (see the bibliography) discusses this problem more fully. In essence, the problem is that every terminal manufacturer has decided on a different set of features and ways of accessing these features. What must be done to solve the problem is to specify a set of routines which can be called which isolate these differences, as well as a way of selecting among different sets of such routines as the terminal changes.

Some systems already have a solution to this problem and interfacing the editor to that solution is the best way out. For the most part (such solutions are RARE), the person who writes the editor will effectively create one. As will be mentioned later, the text editor might very well become the de facto solution to the problem. Other programs would merely output to editor buffers and the editor's redisplay code would take care of the rest.

The following set of routines will allow terminal independent I/O for most terminals. They allow full access to the capabilities of TTYs and basic terminals. They will not allow full access to the capabilities of advanced terminals, but they will get you somewhere. Memory mapped terminals usually use a totally different I/O package anyway and so they are not considered.

Basic I/O:

```
GetChar(Character <r>)
```



```

PutChar(Character)
InputWaiting(Number <r>)
Init(Terminal Type)
Fini

```

The first three routines are capable of handling all input and output associated with a full duplex stream device. End of record marks (e.g., new lines) are transmitted as characters. The first two routines get and put raw characters (no translation or checking of any sort is done) and the other one tells you of the state of the buffer. InputWaiting tells you if the user has typed anything that you haven't read yet. If he has, you can read it before calling the rdisplay. If the input is coming from a file, InputWaiting will tell you the number of characters left in the file. This interface is a general stream oriented interface. These routines update the internally known cursor position to correspond to the new one (i.e., increment by one for the most part on output). Init sets the terminal type and initializes the terminal to a reasonable state (e.g., do not echo input). Fini undoes whatever init did so as to leave the terminal in some reasonable state for general system use (e.g., not raw I/O, echo input, etc.)

Basic Terminal Control:

```

MoveCursor(x,y)
CLEOL
CLEOS

```

MoveCursor knows where the cursor is and figures out the fastest way of getting it to (x,y). CLEOL sends a command to the terminal to clear from the current position to the end of the line and CLEOS clears to the end of the screen.

Advanced Terminal Control:

```

Insert(String)
Delete(Number)
InsertLines(Number)
DeleteLines(Number)

```

Insert takes String and figures out the most reasonable way of inserting it. Delete deletes characters on the current line. The Insert/Delete Lines routines deal with lines on the screen. In all cases, Number can be either positive or negative and a positive number signifies to the right or below of the cursor, respectively.

3.3.6 Echo Negotiation

Echo negotiation was devised for the Multics system and is a protocol for use by multi-node networks which can cut down on response time by reducing communications overhead. It is useful in an environment where the user's terminal is a one node and the computer which is running the text editor is at another. In such an environment, it can take a long time to send a character back and forth (and it takes nearly the same time to send many).

Echo negotiation can only be used when the point is at the end of a line. The editor can download the front end processor (the node closest to the terminal) with a list of approved characters. As long as the user types only those characters and does not reach the end of a screen line (necessitating a

wrap), the front end can safely echo the input characters to the terminal and buffer the input text. When any non-approved character is typed (or the line fills up), the editor is invoked to process the echoed text (the number of already echoed characters is returned to the editor) and the additional character. See section 3.6.2, page 27 to see how this protocol affects the redisplay algorithm.

3.4 Approaches to Redisplay Schemes

There have been two major approaches to performing redisplay. The first is for the routines which are invoked by the user to tell the redisplay code exactly what they did (e.g., "I deleted 5 characters from here"). This approach is not a very clean one and it is prone to error. This is an especially important consideration because we would like to encourage novice users to write their own commands. The extra effort of getting the redisplay correct might make this an impractical goal.

The second approach has been to have the redisplay know nothing about what has occurred. It must rescan the buffer and decide for itself what has and has not changed. This process requires a copy of the screen and can be expensive in cpu time. This algorithm will be presented first because of its relative simplicity.

There is a compromise between these two approaches which seems to solve all of the problems. This compromise is to have the memory management software communicate with the redisplay software. User routines know nothing of this communication and cannot cause bugs in it. On the other hand, the cpu time require for a redisplay is somewhat reduced and is more spread out and so it is not as noticeable. Extra memory is required to handle the communication, but in some cases, the screen representation can be discarded and so the net result could be a memory gain. It is this compromise that is the heart of the "modern" redisplay and it is the other one to be presented.

3.5 The Framer

The framer is the part of the redisplay that decides what will appear on your screen. In the stable state, there are two different approaches used.

First, the TopOfScreen and BottomOfScreen marks are kept around. As long as the point stays within these marks, we expect that the point will remain on the screen. Thus, the top of the screen can be assumed to be in the proper place and the redisplay algorithm can be started directly. If it does not (the redisplay code detects this error and generates a FramerError), the framer runs again, but uses the next approach.

Second, if the point is outside of the screen marks, it is simplest to assume that the entire screen will be changed. Thus, the framer wants to recenter the point on the screen. It can start by counting back $\langle \text{screen height} \rangle / 2$ lines. Assuming that there are no wrapped lines, this method would work fine. At this point, the framer checks this assumption (that there are no wrapped lines) by counting forward character by character, keeping track of how many lines are actually used along with the intermediate results. If there are no wrapped lines, the new guess will work fine. If there are wrapped lines, it will look at the intermediate results and decide how many lines to throw away to leave you approximately centered. If the advanced redisplay algorithm is used, these intermediate results should be recorded as they might be needed.

If all the lines have to be thrown away (i.e., the current line is VERY long), the third and most

desperate mode must be used. Here, the framer figures out, character by character, where each character on the current line is. It then decides how many characters to move back before starting the redisplay, while staying within the same line.

3.6 Redisplay Algorithms

Here are presented the two major redisplay algorithms and an discussion of how to adapt these algorithms for memory mapped terminals. These algorithms will not go into every detail (or even most of them) as doing so would inundate the description with too much detail. This detail is discussed in later sections.

3.6.1 The Basic Algorithm

```

call Framer;
                                /* TopOfScreen is a mark returned by
                                the framer */
BufLoc = TopOfScreen;
                                /* loop over the whole screen */
do Row=1 while(Row <= HeightOfScreen);
  do Col=1 while(Col <= WidthOfScreen);

                                /* found a NewLine char */
    if Buffer(BufLoc)=NewLine
      then do;
                                /* is the rest of the line blank? */
        do i=Col to WidthOfScreen;
          if Screen(i,Row) != " "
            then do;
                                /* if not, make it so by
                                sending a CLEOL at the
                                non-blank */
                                call MoveCursor(i,Row);
                                call CLEOL;
                                do j=i to WidthOfScreen;
                                  Screen(j,Row)=" ";
                                end;
                                leave;
          end;
        end;
        BufLoc = BufLoc + 1;
                                /* move to next line */
        Row = Row + 1;
        Col = 1;
        leave;
      end;

                                /* no NewLine, so has there been a
                                change in the buffer? */
    if Screen(Col,Row) != Buffer(BufLoc)
      then do;
                                /* if so, change the screen

```

```

                                to match */
                                call MoveCursor(Col,Row);
                                call PutChar(Buffer(BufLoc));
                                Screen(Col,Row)=Buffer(BufLoc);
                                end;
                                BufLoc = BufLoc + 1;
                                Col = Col + 1;
                                /* save the (x,y) of the point so
                                   that we can put the cursor there later */
                                if BufLoc=Point
                                then do;
                                    PointX = Col;
                                    PointY = Row;
                                end;
                                end;
                                Row = Row + 1;
                                Col = 1;
                                end;
                                /* framer missed--it almost never happens */
                                if BufLoc < Point
                                then call FramerError;
                                EndOfScreen = BufLoc;
                                call MoveCursor(PointX,PointY);

```

This algorithm is quite straightforward. It first calls the framer to match the top of the screen with some point in the buffer. It then iterates through the buffer and the screen simultaneously, matching characters as it goes. As long as the character on the screen matches the character in the buffer, no action is taken. When there is a discrepancy, the cursor is moved to that position by means of the MoveCursor routine, the changed character is printed, and the screen array is updated. If the line gets to be too long, it is wrapped automatically. If a NewLine character is encountered, the rest of the line is checked to make sure that it is all blanks. If not, blanks are put there. Finally, a note is made of where in the buffer the end of the screen falls.

This is your basic, garden variety redisplay algorithm. It will work on any terminal that supports cursor positioning (the CLEOL call can be faked by sending spaces). It will work quite well on anything running at 480 cps or over. Its only memory requirements are an array large enough to hold the screen (typically 1920 characters). The only interaction between the redisplay algorithm and the memory management system is two marks. Finally, it is not told anything about what changes were made and so it figures everything out for itself each time it is called. There can thus be a cpu time penalty associated with this algorithm that might make it slow enough to be painful. The next section describes with an algorithm which gets around this penalty.

A complete redisplay can be generated quite easily using this algorithm. The GenerateNewDisplay routine will set the cursor to home and then clear the screen and the internal screen array. It then calls the incremental redisplay routine. The incremental redisplay routine will simply do its normal job, which in this case implies sending all of the non-blank characters to the terminal. The NewDisplay routine must also remember to send such things as status displays, which are not sent during an ordinary redisplay.

A status display is text that is kept on the screen but is not often changed. For example, the Emacs

status display has the editor name, the mode name, the current buffer name, and the file name displayed on a line near the bottom of the screen. Ordinarily, the redisplay code ignores this section of the screen.

3.6.2 The Advanced Algorithm

The advanced redisplay algorithm serves two vastly different purposes. First, it provides a way of efficiently taking advantage of the insert/delete line/character functions which are supplied with some terminals. Second, it provides a low cpu overhead way of performing a redisplay on basic terminals.

The basic idea used by this algorithm is to assign a unique id to each buffer line that appears on the screen. Note that a buffer line can take up more than one line on the screen by wrapping. Just to make sure that the definitions are clear, here they are: a buffer line (BufferLine) is either the text between two newline characters (in the buffer gap memory management scheme) or the text in one element of the line list (in the linked line scheme). A screen line (ScreenLine) is a horizontal row of characters on the user's display.

The unique id can be in any form. One method is to use a 32 bit counter and increment it each time any line is changed. After the change is made, the line is assigned the current value of the counter. If it is changed again, it gets the new value of the counter. The assignment can be made in an otherwise unused part of the header (for linked line) or in a special mark (for buffer gap). In a linked line scheme, the pointer to the line can serve as a unique id.

These unique ids only have to exist for lines that appear on the screen. Thus, the buffer gap scheme only has a few of these special marks that must be maintained. The special marks are placed at the beginning of each line that appears on the screen. They contain a version number for the line as well as the location of the mark.

The memory management scheme is responsible for maintaining this extra information. Thus, it and the redisplay code can interact heavily and the specific redisplay process chosen will affect the internal structure of the memory management scheme.

There are two flags that can be kept by the memory management software which will aid the redisplay process. First is the buffer modified flag. This flag is usually kept anyway so that the editor can detect when the buffer has been modified. (The details of manipulating it were discussed with the interface routines in section 2.1, page 3.) If it has not been set, the redisplay code knows that it generally will not have to do anything except move the cursor. If the point is still on the screen (remember that we have beginning and end of screen marks), its position on the screen can be calculated with much less effort that is required for a full redisplay. If the flag has been set, a full redisplay is required and the flag will be reset (the editor proper ORs this flag in with a private flag (Modifiedp; mentioned in the buffer data structure descriptions) in order to properly remember whether the buffer has been modified).

Another flag (which has not been mentioned before) can significantly reduce redisplay computation in some cases. Assuming that you are located at the end of a BufferLine, it tells you whether or not any operation other than inserting a character has been done. If the flag says not, all that the redisplay has to do is output the one character (after checking for wrap, etc.). A significant

amount of time can be saved this way, but it is most useful with a negotiated echo protocol (see section 3.3.6, page 23). The exact interface to this flag will not be defined.

The redisplay algorithm itself starts by trying to find a match between the BufferLines and the ScreenLines by using unique ids. The unique ids are compared, line by line. If they match, no work needs to be done and the redisplay proceeds to the next line. If they don't, it can be for one of three reasons:

- An additional line (or lines) was inserted between the two ScreenLines. This condition is detected by comparing the ScreenLine unique id with all of the BufferLine's unique ids and finding a match. (Remember that the ScreenLines are what the BufferLines were one redisplay iteration ago.) We thus have the situation where we used to have A,B and now have A,<junk>,B. Clearly, the most reasonable assumption is that <junk> has been inserted. We thus count how big <junk> (the framer has already calculated this information) is and tell the terminal to insert the appropriate number of lines. (Before you do this, however, you must first delete the same number of lines from the end of the window in order to keep from losing the text at the bottom of the screen.)
- A line (or lines) was deleted. This is detected by comparing the BufferLine unique id with all of the ScreenLine's unique ids and finding a match. We thus have A,B,C becoming A,C. We delete the appropriate number of lines and then insert them again at the bottom of the window.
- The line was modified. This is detected by not finding either of the above matches. At this point, we switch to intra-line work and do the following:
 - *Do a string compare starting from the beginning of each line (the BufferLine and the ScreenLine) and see how much they have in common. (If this says the whole line matches, no more work has to be done.) For example, if the ScreenLine is "abcdef" and the BufferLine is "abxdef", they have two characters in common from the start.
 - *Do the same thing starting from the end. The example strings have three characters in common from the end.
 - *Compare the line lengths. If the two lines are the same length, you only need to rewrite the changed part (e.g., two characters were interchanged). In the example strings, the lengths are the same (6). This optimization can be done even on a basic terminal. If the two lines are not the same length (for example, the ScreenLine is "abcdef" and the BufferLine is "abxyzdef"), rewrite as much of the portion between the common text sections as possible ("x") and then either insert or delete the required number of characters (in this case, insert two blanks) and finish writing the modified text ("yz"). Remember that if there is no common text at the end and the BufferLine is shorter than the ScreenLine, a C1.EOL call is appropriate.
 - *Wrapped lines can pose a problem. There may be no end common text, and yet an insert or delete character operation might be the appropriate one. (If the screen

width is six characters, the ScreenLine is "abcdef", and the BufferLine is "abcxdef". Here, the BufferLine will ultimately become two ScreenLines: "abcxde" and "f".) This case is detected by having no end common portion and noticing that the line wraps. A more complicated matching process can detect the situation and appropriate action can be taken.

3.6.3 Memory Mapped

Redisplay for memory mapped terminals boils down to one of three cases. Each case is relatively simple.

1. Reading from and writing to the screen cause flicker. The solution is to use the basic terminal redisplay scheme.
2. Reading does not cause flicker but writing does. The solution is to use the basic terminal redisplay scheme, but use the actual screen memory for storing the screen array.
3. Neither reading or writing cause flicker. On each redisplay cycle, merely copy the buffer into screen memory, not forgetting to process new lines, etc., as needed.

3.7 Other Details

There are a number of other details that must be carefully watched when writing redisplays. None of them are particularly worrisome in themselves, but they collectively clutter the algorithms a great deal. The problems that they pose will be described and they are each simple enough that specific solution algorithms are not required.

3.7.1 Tabs

It helps to think of a tab character in a buffer as a cursor control command saying, "think of me a N blanks, where N is the number of columns to the next tab stop." Thus, whenever you see a tab you want to figure out what N is, and then check to see that the next N columns are blanks, increment the cursor by N, etc. Tab stops can be set in an array (for arbitrary placement of tabs) or set every C columns. In a one origin numbering system, tabs set every C columns are set at positions 1, C+1, 2C+1, 3C+1, ... For example, when C=8, tabs are in columns 1, 9, 17, 25, 33, etc. Again, assuming a one-origin system, the equation for N is:

$$N = C - \text{mod}(X-1, C)$$

(X is the column position.)

3.7.2 Control Characters

In general, only the new line character(s) and tabs are interpreted; other control characters are displayed in some reasonable printing representation. One popular representation is "↑" followed by the character whose ASCII value is <control char> + 64. The character control-a is thus printed as ↑A. (The ASCII DEL character, 127, can be printed as ↑?) This convention has been followed in this thesis. When displaying control characters, you must remember that while the character itself is

only one character, it displays in a two character wide sequence. In addition, it is the actual displayed sequence that is stored in the screen array (e.g., "†" and "A", not "†A"). Care must be taken to insure that control characters can wrap properly across line boundaries (e.g., the "†" is not displayed at the end of one line with the "A" at the beginning of the next).

3.7.3 End of the Buffer

If the entire buffer fits on the screen, you will run out of buffer before you run out of screen. Thus, whenever BufLoc is incremented, a check should be made against the buffer length. If you do run out of buffer, remember to finish blanking the rest of the screen if it needs it.

3.7.4 Between Line Breakout

The redisplay process does not have to run to completion before editing resumes. Instead, it can get to a convenient spot (in the basic algorithm, almost any spot will do; in advanced algorithm, stop after finishing a line) and check the input buffers. If more input has arrived, it can abort the redisplay and process the input. Remember that the purpose of redisplay is to provide feedback to the user. If he has already typed something, he does not need feedback immediately. (However, if you can give it to him in a way that does not slow him up, do so.)

3.7.5 Proportional Spacing and Multiple Fonts

Displaying text in a proportional spaced font is not too difficult. Instead of assuming that each character has a width of one, the width can vary and it must be looked up each time it is needed.

Displaying multiple fonts implies receiving a command to switch fonts at some time during the redisplay process. These commands can be stored in the buffer (in which case like NewLines they are interpreted and not displayed just) or in some other structure.

3.7.6 Multiple Windows

There is a database somewhere which describes what windows (i.e., what part of which buffers) are to appear on the screen. One way to perform redisplay with multiple windows is to call the incremental redisplay routine and pass it as an argument each window descriptor in turn. Another way is more suitable for use with the advanced algorithm and it involves having a separate descriptor for each line of the display (i.e., the same database sorted backwards as well). This descriptor tells you where to get each line from.

If a row of dashes ("-----") or any other character string is used as a visual separator between windows, it can be implemented as an additional buffer/window combination and no special casing is required for the redisplay code.

4. The Command Loop

This command loop is the part of the editor that actually implements the logic of the editor. It is responsible for reading in commands, executing them, and "printing" the results. In the process of executing them, it must accept arguments and bind the input characters to functions. This chapter will discuss the command loop. It will also discuss some distantly related issues: the tradeoffs between kill buffers and an undo function, the provisions for recovering from errors, and considerations for selecting implementation languages.

4.1 Basic Loop: Read, Eval, Print

The basic loop is:

```
do while(TRUE);
  call GetChar(Char);
  call Eval(Char);
  if abort
    then leave;
  if InputWaiting() = 0
    then call IncrementalRedisplay;
end;
```

Note the two details that have been added to what was mentioned in the section heading. First, an abort flag is checked to see whether we are supposed to exit the edit session. This flag is set by the Eval routine. Eval works by invoking a function which was specified by the input character. This function's only result is the change in the state of the editor (e.g., an "x" has been inserted). The "printing" (actually, an incremental redisplay to the screen) is done only if the user has not typed in anything more to be processed.

4.1.1 The Philosophy Behind the Basic Loop

The basic loop as described puts the fewest restrictions on the user interface that can be managed. Each character, in its raw form, is mapped to a procedure which is in turn evaluated. Any arbitrary syntax and semantics can be implemented with this base.

In theory, a syntax of commands being words (e.g., "delete", "move", etc.) could be implemented in this structure by having either a large number of dispatch tables (and thus implementing a symbol state table architecture) or a procedure which parses the syntax of the command via conditional statements. For reasons which will be stated, this syntax is not generally implemented.

Consider the thought that every character that is typed at the keyboard causes a function to be executed. The first conclusion that results is that it is silly to type "insert x" or anything like that when you want "x" to be inserted. As this is a very common operation, it makes more sense to bind the key "x" to the InsertX function. (Actually, it is probably bound to SelfInsert, a function which looks at how it was invoked--the input character--to determine what to insert).

Now, all of the straight, printing, ASCII characters have been taken and bound to SelfInsert. (While there are a large number of special characters that are not often typed, leaving them in consideration does not materially affect the conclusions.) The remaining things that can be entered

from an ASCII keyboard are the control characters, the delete key, and the break key. These could be bound to functions that implement a complex syntax, but why bother? It is not too difficult to learn even a large number of key bindings, so let us bind the control keys directly to useful functions. For example, \uparrow F could be ForwardCharacter, \uparrow D could be DeleteCharacter, etc.

33 functions are not enough for even the commonly used functions. Thus, some of the keys should be bound to functions which rebind the dispatch table. For each of these rebinding functions, 128 new functions are made available (there is no reason for the printing characters in them to be bound to SelfInsert). Note that the break key is not used in this scheme as it is hard to work with (it does not have an ASCII value).

Thus, even though we began with a structure for the command loop that did not impose any constraints on the syntax of commands (and thus was as general as possible), we arrived at a specific syntax for commands. This syntax is to bind the printing characters to SelfInsert, bind the control characters to a mixture of useful functions and rebinders, and to have about three or four alternate dispatch tables (enough to supply many hundreds of commands). Thus, commands are rarely more than two keystrokes long. The price that is paid for this brevity is a longer lead time in learning to use the editor effectively.

(Note that most of the increased lead time in learning the editor is NOT from the brief commands, but because there are more commands to learn. Given a "conventional" editor (e.g. DEC's SOS) and an equivalent subset of an Emacs-type editor, novice users will learn the subset of the Emacs-type editor faster.)

4.2 Error Recovery

Errors come in two flavors. There are internal errors which are in the editor itself (e.g., a subscript out of range) and external errors which are caused by the user (e.g., attempt to delete off the end of the buffer). There is also a non-error, the normal exit, which will be treated as an error in this discussion. These errors will, in general, be indicated both from within the editor and from the outside world (the operating system).

The first category to be considered will be internal errors. These errors cause an immediate exit to the operating system with no questions asked and no delays tolerated. They will be internally generated by such things as arithmetic overflows and bad subscripts. (While the editor might catch and process some of these, it will not in general process them all.) They can also be generated externally and often are (e.g., process switching). The factor in common is that they are unpredictable and the state of the editor should remain exactly intact. The user should also be able to signal such an error to abort out of the editor. He might want to do this because of a problem with the editor itself (e.g., infinite loop) or because he wants to do something else. This signalling is usually done with the help of the operating system. In any case, the precise state of the editor should be retained so that it can be resumed exactly where it left off. Most operating systems have some facility for doing this; they differ principally in the freedom of action that they allow before losing the state. This freedom ranges from nothing to doing arbitrarily many other things.

At the user's discretion, the editor should be restartable either from exactly where it left off or at a safe restart point. This point is ordinarily a portion of the editor which recovers the buffers and other current state and then resumes the command loop.

External errors are principally user errors. The action ordinarily taken is the display of an error message and a return to command level. The implementation of this level of recovery is built in to the procedures which implement the commands.

There is a variation of external errors which are generated manually by the user. Typically, these involve backing out of an undesired state (e.g., the unwanted invoking of a dispatch table rebinding or aborting an undesired argument). The bell character (ASCII \uparrow G) has often been used for this purpose. In this case, the procedures will know that a bell has been typed and will implement the backout protocol.

Finally, provisions to exit the editor must be made. This is ordinarily by means of an abort flag of some sort as can be seen in the previous code fragment. Note that various other uses might be multiplexed onto this abort flag, signifying varying levels of "exiting." For example, one level could be used by buffer switching in order to rebind the dispatch tables (see the section on later in this chapter).

Ordinary exiting involves several types of processing. The editor might ask the user what to do with buffers that have been modified but not written out. If, as is ordinarily assumed, the state of the editor is preserved across invocations, the state must be saved. If not, it must be sure that all memory is deallocated. Finally, the user's environment should be restored as it was found. This implies such varied things as cleaning up the stack, closing files, deallocating unneeded storage, and resetting terminal parameters.

4.3 Arguments

Arguments are specified by the user to modify the behavior of a function. The Emacs argument mechanism will be described as an example of three diverse ways in which arguments are obtained.

There are three standard argument types. First are prefix arguments. These are invoked by a string of functions (which are in turn invoked by characters typed before the "actual" command) and are an example of using the key/function binding to implement a more complicated syntax. Next are string arguments. When obtaining a string argument, the editor is invoked recursively on an argument buffer and upon return from the recursive invocation the contents of that buffer are given to the requesting procedure. Last are positional arguments. These are the internal variables of the editor.

4.3.1 Prefix Arguments

Prefix arguments are entered before the command whose behavior they are modifying, thus, their interpretation must not depend upon the command. Emacs limits these to numeric values.

Ordinarily, commands will have an internal variable available to them named something like "argument" and it will have a value of one. Prefix arguments allow the user to change that value to any other positive or negative integer.

Arguments are used for two different purposes. First is to specify a repeat count for a command. Thus, $\langle 12 \rangle \uparrow F$ would go forward twelve characters (assume the $\uparrow F$ key is bound to the ForwardChar function). The other use is to tell a command to use an alternate value for a parameter. If

FillParagraph was bound to $\uparrow P$, then $\langle 65 \rangle \uparrow P$ might say to, for this time only, use 65 as the desired width of the paragraph (the right margin) after it is filled. Alternatively, it might say to reset the default value of the right hand margin to 65 and then use that value. It is useful to provide a predicate to allow procedures to determine whether an argument has been given. This allows them to differentiate the default argument of one from the user entering one as the argument value.

Emacs uses $\uparrow U$ as the UniversalArgument function. It can be used in either of two ways. $\uparrow U \uparrow F$ means to go forward four characters. Adding another $\uparrow U$ means to multiply the current argument by four. Thus, $\uparrow U \uparrow U \uparrow U \uparrow F$ means to go forward 64 characters. The factor of four was selected because five is too large (1, 5, 25, 125 goes up too fast) and, while three might have better spacing (1, 3, 9, 27, 81, 243), the powers of four are known by all people who are likely to be around computers.

The other use is more complicated. $\uparrow U 12 \uparrow F$ means to go forward twelve characters. $\uparrow U -147 \uparrow A$ means to give $\uparrow A$ an argument of -147. The $\uparrow U$ in this case serves as an "escape" to logically rebind the 0-9 and - keys.

On some terminals, there are two sets of numeric keys (one set that sends the ASCII "0" - "9" codes and another that is labeled with digits but sends different codes) to generate "numbers" than simply sending the appropriate ASCII codes. In this case, these "other numbers" can be bound directly to argument generating functions and the initial $\uparrow U$ is not needed.

4.3.2 String Arguments

String arguments are specifically requested by a procedure. A prompt is displayed and the user enters the value of the argument. The procedure uses this value in any way it desires.

One way to implement such a way of entering arguments is to create an argument buffer in a new window, display a prompt, and call the editor recursively with that as the current buffer. By following this scheme, the full power of the editor is available to correct typing mistakes or otherwise make the entry process easier.

When implementing any argument entry scheme, there are three things to take into account. First, the key or key sequence used to indicate that the entry process is over should be able to vary depending upon who is asking for the argument. $\uparrow M \langle \text{cr} \rangle$ and $\uparrow [\langle \text{esc} \rangle$ are both commonly used as delimiters. Second, there should be a clean way to abort out of the argument entry process ($\uparrow G$ is commonly used for this purpose). In this case, the calling procedure should be told about the abort in order for it to terminate gracefully. (Most of the routines that ask for arguments do all of the asking at once and then proceed to do a large amount of work (e.g., ReadFile). Thus, aborting out of the argument entry process effectively aborts out of the command. Aborting cannot be done cleanly if commands are written to get an argument, do some work, get another argument, etc.) Finally, null arguments (the user enters only the delimiter character) can be used to cut down on typing errors if the procedures supply some reasonable default values.

Here are some examples of using string arguments:

SearchString: Ask for a string and look for it in the buffer. If the user enters a null string, use the same string that he searched for before.

ReadFile: Ask for a string and, using it as a filename, read the file into the buffer. If the user enters a null string, use the current filename associated with the buffer.

ChangeBuffer: Ask for a string and, using it as a buffername, make that buffer the current one. If the user enters a null string, use the buffer that he was in last (i.e., the one that he was in before the one that he is in now).

Note that `SearchString` typically uses `↑` (`<esc>`) as the delimiter while `ReadFile` and `ChangeBuffer` typically use `↑M` (`<cr>`). In order to help the user, it is nice to automatically remind him which delimiter is being asked for. Here are some example prompts:

```
Search String(<esc>):
Input File Name(<cr>) (Default is >u>fin>test):
Buffer Name(<cr>) (Default is foo):
```

Note that some prompts helped the user by reminding him of the default value.

While all of the examples asked for and wanted a character string, this might not always be the case. It is quite practical to use this method to enter numeric values. The requesting procedure merely has to convert the read-in character string to a numeric value.

4.3.3 Positional Arguments

Positional arguments are not directly specifiable by the user. They are the internal variables that are used in the editor. Such variables include both those required by the editor (e.g., the length of the buffer, the locations of the point and the mark, etc.) and those which have a specialized purpose (e.g., the current value of the right hand margin, the tab spacing, etc.).

Often these values are used in unusual ways. For example, the horizontal position (column) of the point can often be a more pleasant way of specifying a value than entering a number. The user can indicate that "this is where I want the right margin to be" instead of having to count characters to get a number. The user indicates this value by using other commands (e.g., `ForwardChar`, `ForwardWord`) to move the point to the desired location. See also section 5.2, page 42 for information about how graphical input devices (mice, tablets, touch sensitive displays) affect positional arguments.

4.4 Rebinding

Rebinding is a name for the act of changing at run time what a key or procedure does. The distinction between the two (keys and functions) is important. Changing the binding of a key means that when that key is typed, the new procedure (the one that is now bound to the key) will be executed instead of the old one. Changing the binding of a procedure means that whenever that procedure is invoked, the new version will be executed instead of the old one. This change affects not only any keys bound to that procedure but also any internal references to it.

There are two levels of rebinding functions. Level I rebinding is when the new procedure must be known before invoking the editor. Level II rebinding is when the new procedure can be defined after the editor is invoked. Unless otherwise stated, level II rebinding is assumed.

To a first approximation, editors that are written in compiled languages (e.g., PL/1) can only change the key bindings and interpreted editors (those written in, say, Lisp) can change both bindings. Dynamic linking, however, allows both bindings to change in compiled editors and so this distinction is not always a proper one to make.

4.4.1 Rebinding Keys

The process of key rebinding is a relatively simple one and it is done essentially the same way in all implementations. A set of dispatch tables is used to map keys (represented by their ASCII values) to their respective functions.

In languages such as Lisp and PL/1, the table can contain the procedures themselves. In less powerful languages such as Fortran and Pascal, the dispatch table branches to a different part of the same routine that contains the table. There, the procedure call is made. In languages that supply it, a case statement can be used instead of the n-way branch.

None of these command procedures have any formal parameters, and so they can all be invoked with the same calling sequence. Thus, the Lisp and PL/1 direct invocations can work properly. Note also that simple commands do not have to have a separate procedure assigned to them, but the code to execute them can be placed in-line in place of a call (where the case statement equivalent is used). Doing this substitution loses some potential flexibility.

4.4.2 Rebinding Functions

Level II function rebinding is ordinarily a language-supplied feature and so it will not be discussed in depth. Two comments will, however, be made on how to simulate it.

If the underlying operating system has dynamic linking (e.g., Multics), a procedure may be rebound at run time. Dynamic linking is a way of linking procedures together in which the actual link is not made until the procedure is about to be executed. At that time, the procedure is located in the file system and brought into memory. The link may either be left alone, in which case the next call will have the procedure re-located (a relatively expensive process) or it may be snapped. Snapping a link implies converting the general call instruction (which is kept in a special, writable part of the program) into a call instruction to the appropriate address. If a link is snapped, it must be explicitly un-snapped before any re-locating is done.

If the operating system does not support dynamic linking, the editor writer might choose to simulate it manually. Such a process is complex and some thought will have to be given to exactly how desirable rebinding functions is. The process is tantamount to explicit overlaying.

This all has a straightforward bearing on rebinding functions. Rebinding a function involves changing the definition of the procedure that is invoked by referencing it. What has been discussed are ways of changing such a procedure definition. Note that if the code to execute a function is inserted in-line in the basic editor, it cannot be rebound by any of these methods.

If dynamic linking is not available and is unfeasible to simulate, there is still one way out. This way will only provide level I rebinding. Instead of just using one dispatch table which indicates a procedure to be called directly, use two. The first table maps from keys to the operation to be

performed (e.g., `↑F` is mapped to moving forward one character). The second table maps from the operation to be performed to a procedure to perform it (e.g., moving forward one character is mapped to `ForwardChar`).

4.5 Modes

Modes are collections of rebindings which are done all at once. They can either be done automatically or can be explicitly asked for by the user.

An example of an automatically loaded mode might be PL/1 mode. This mode will automatically be loaded whenever a file whose name ends in ".pl1" is read into a buffer. Such a mode might do several things. It might rebind the internal variable that identifies which characters are legal in tokens (i.e., variable names) to also include the "\$" and underscore characters which can occur within PL/1 names. This change would make the `ForwardWord` function treat a PL/1 variable name as a word. The mode might also rebind the ";" key to be an electric semicolon (i.e., finishing one statement would cause it to automatically indent properly for the next one).

The process of autoloading is related to automatically loaded modes. The trigger is the main difference. In autoloading, the trigger is completely internal. An example could be the set of S-expression hacking commands. Although they are defined at all times, the code for them is not necessarily a part of the editor. Instead, when any of the commands is invoked, they are autoloaded into the editor and the command is executed.

An example of a user requested mode would be auto fill mode. This mode rebinds the space character to one that checks to see if you are typing past the right margin. If you are, it breaks the line up to fit within the right margin. It also inserts the space.

A printing terminal mode would use function rebinding. It would be loaded automatically whenever the editor is used from a printing terminal instead of a display. It might rebind the `SelfInsert` function (which is used by all of the 95 printing keys) to one that prints the character that it is inserting on the terminal (and then inserts it). In this case the definition of the function changed and so function rebinding is called for. Note that this change is global over all buffers and so it is not readily simulatable by changing the bindings of keys to operations.

The function rebindings that are commonly done by an editor are known in advance and so they can be done by any implementation (see the preceding section for a discussion of the difficulties involved in function rebinding). Fully dynamic rebinding (the new definition of the procedure is not known until run time) is desirable for several reasons.

- Debugging is greatly eased if the trial-and-error cycle time is reduced by not having to compile and link the whole editor each time. Instead, only one function has to be recompiled and linked. (In languages such as Lisp, it is more accurate to say compiled/linked as the two operations are synonymous.)
- Space savings are achieved if unneeded modes and autoloaded single functions are not brought into memory until asked for.
- If the editor is implemented in an interpreted language (see the next section) users can

develop their own functions relatively easily. Such "sideline" development is advantageous because it allows many people to develop useful code and so the editor can be specialized in many more ways than any reasonable support group could ever implement on their own. It also encourages tailoring the editor to a user's own taste and so his productivity is enhanced.

4.5.1 Implementing Modes

Modes are on a per buffer basis and so provision must be made for changing these bindings as buffers are switched. The general technique for doing this is to have a set of default bindings and a set of current ones. When a buffer switch is made, the default bindings are copied to the current ones and then a series of procedures are run which modify the set of current bindings to be the correct ones for the modes that are active on this buffer.

A different approach would be to have a separate environment for each buffer which is created with the buffer, is modified as modes are added, and is never thrown away. This approach leads to efficiency problems because of the large amount of storage overhead associated with each buffer.

Sidenote Calculation: Assume that there are two dispatch tables of 128 commands each and that each entry is four bytes (big enough for an address). This leads to 1K bytes just for the dispatch tables per buffer. In addition, you have another 1K bytes for a default table to use when creating a new buffer. With a current/default dispatch table scheme, you have 2K bytes per editor and so you are always as efficient and better in the case where you have more than one buffer. Procedural storage overhead is essentially the same. In one case, you invoke the state building procedure once (but in general cannot undefine the procedure) and in the other case, you invoke it with each buffer switch. It does, on the other hand, take longer to switch buffers but the incremental time is usually minimal.

There is an important flexibility tradeoff. With a mode list and the associated default/current dispatch tables, it is possible to remove a mode from a buffer. If each buffer has its own dispatch table which is incrementally changed whenever a new mode is added, it is not generally possible to undo such changes. Note that while the dispatch tables were used as an example, it is by no means the only variable whose value may change on a per-buffer basis.

4.6 Kill and Undo

An Emacs maintains a kill ring which is a place where all significant chunks of deleted text get placed. (Those deleted with C-d and do not get saved.) There are commands to push and pop things from the current spot in the ring and to rotate the ring so that different text is at the current spot. Typically, a maximum of ten or so items are kept in the kill ring.

Moves and copies of text are done with this ring. Thus, there is a mechanism by which the user can recover accidentally deleted text. This type of error is the most harmful one that can occur as it involves losing information.

The Interlisp system (and others) provides a more general undo facility. Invoking this facility will cause the system to "undo" whatever it was that you just did (for one command only; a second

"undo" will undo the first one). In order to implement this facility, the system must keep track of everything that you do and what its effects were.

While this general purpose facility has good applications, it is not clear that a text editor is one of them. There are three basic areas where undo applies to text editing. These are: moving around in text, deleting text, and file i/o. The Emacs approach and the undo approach will be compared for each of these.

Moving around in text is simply solving the problem "I am at x and I want to be at y." The Emacs solution involves translating this difference into a sequence of commands to move the point from z to y. If a mistake is made in the process of implementing the solution, the problem is merely restated to "I am at x' and I want to be at y" and it is re-solved. The undo solution differs by detecting the error (i.e., deviation from the intended solution), saying "undo" to put you back on the original path, and proceeding. Ordinarily this difference in the two solutions is not very great.

If the accidentally typed command is one that moves you a great deal (e.g., move to the beginning of the buffer), it is not always easy to recover with the Emacs solution because you might not remember exactly where you were. Emacs solves this by having the large movement commands set the mark to where you were. Thus, an interchange point and mark sequence will recover from the error.

The undo actually helps less in the text deletion case. There, the "canonical" undo will only recover the last command and, hence, the last delete operation. There is no provision for deleting something, moving somewhere else, and undeleting it. Nor is there a provision for recording multiple deletions. Thus, the Emacs approach is more flexible.

Finally is the case of file i/o. Different implementations of Emacs will do different things but the basic idea is to let the user do what he wants. Obvious things will be checked (the file was modified by someone else since it was read in, for example) and such things as deletions will be double checked with the user but no recovery will be provided. On the other hand, not all systems can support the overhead of the multiple copies of a file that would be required by undo, nor are there always ways to manage these extra files conveniently. (The DEC TOPS-20 operating system does do a reasonable job at this, but it is far from perfect.)

The basic conclusion is that while an undo facility is nice, it is not all that useful in the context of an Emacs type text editor.

4.7 Implementation Languages

The language that the editor is implemented in can greatly affect the ease of writing, maintaining, and extending it. Some brief comments will be made about several classes of programming languages which might be considered as implementation languages.

4.7.1 TECO

(This discussion refers to MIT TECO and not the TECO which is supported by DEC on several of its machines. MIT TECO is much more powerful.) TECO is a text editor. Its command language is so powerful that it is usable to write other programs in. It is tailored for writing text applications and

so would seem a good choice. It has two major problems:

- It is the only language less readable than APL. A listing of a TECO program more resembles transmission line noise than readable text. Thus, maintenance is a problem.
- Its only implementation is on the PDP-10/DEC 20 series of computers. Implementations on other machines involve asking the question of what you write the TECO in.

4.7.2 Sine

Sine is a Lisp-like language tailored for text applications. Its only implementation to date is on Interdata 7/32 (or Perkin-Elmer 3200) minicomputers running the MagicSix operating system developed at MIT. It is interesting because it is a language tailored for implementing editors. It is an example of an "ideal" implementation language. [Anderson] discusses this language in detail.

Sine is composed of two parts. Sine source code is assembled into a compact format. This object code is then interpreted. It allows function rebinding and other such niceties and the interpreter implements such things as memory management and screen redisplay automatically. Thus, the resulting editor is nicely structured, with "irrelevant" details hidden away.

4.7.3 Lisp

Lisp is probably the best choice, if it is available. The Lisp must, however, have string operations in order to run with any efficiency. It is best suited for the linked list form of memory management because of its view of memory management. Lisp provides a nice interpretive language for escaping into to easily write complicated editing macros. It also is quite readable and maintainable. It also provides function rebinding. Some Lisps have compilers whose code can run very fast, so speed need not be a problem.

4.7.4 PL/1, C, etc.

PL/1, C, and other such "systems languages" are widely available in reasonably efficient implementations. They allow the straightforward manipulation of complicated data structures and yet remain generally readable. They specifically support containment of detail by independently compiling several related routines and their internal data structures.

As a specific example of the latter, it is possible to write a buffer management abstraction in which the only visible parts are the entry points. The specific method chosen to represent the buffer remains well hidden.

4.7.5 Fortran, Pascal, etc.

Fortran, Pascal, and other such languages are the least acceptable (except, of course, for assembler). In general, one must either simulate a missing basic feature (e.g. Fortran and If-Then-Else) or circumvent a "feature" (e.g., Pascal and lack of multiple entry points to procedures) in order to do useful work in such languages.

5. User Interface Hardware

The only way for a user to interact with the text editor specifically or the containing operating system generally is by means of the keyboard/screen combination. The chapter on Incremental Redisplay discusses the use of the screen in detail. This discussion is on the keyboard part of the combination.

5.1 Keyboards

The keyboard is the primary means of interacting with the system. In most cases, it is the only way of doing so. Many thousands of characters will be entered in the course of a normal working session. Thus, the keyboard should be tailored for the ease of typing characters. While the previous statement might seem trite, there are a large number of keyboards on the market which are not very good at all for entering characters. Here is a discussion of the various keyboard features and why they are or are not desirable:

N-KEY ROLLOVER is a highly desirable feature. Having it means that you don't have to let go of one key before striking the next. The codes for the keys that you did strike will be sent out only once and in the proper order. (The "n" means that this rollover operations will occur even though every key on the keyboard has been hit.) The basic premise behind n-key rollover is that you will not hit the same key twice in a row. Instead, you will hit a different key first and the reach for that key will naturally pull your finger off of the initial one. However, the timing requirements are quite loose about exactly when your finger has to come off of the first key. Thus, typing errors are reduced. Note that n-key rollover is of no help in typing double letters. Note also that shift keys and the control key are handled specially and are not subject to rollover.

AUTO-REPEAT has both good and bad sides to it. It is useful on a system which does not supply such things in software but its drawbacks (leaning on a key can be deadly) makes it out of place on a system with a sophisticated editor. (If you want a row of "."s, just type "+U 80 .".)

TOUCH-TYPABILITY is the single most critical feature. It is simply the ability to type the useful characters without moving your fingers from the standard touch-typing position (the "asdf" and "jkl;" keys). As more and more people who use keyboards are touch typists and can thus type at a reasonable clip, they should not be slowed down by having to physically reach their hands out of the basic position. It can take one or two SECONDS to locate and type an out-of-the-way key. (The row above the digits is out-of-the-way, as are numeric key pads and cursor control keys.) One second is from three to ten characters of time (30 - 100 words per minute). Thus, it takes less time in general to type a four or five character command from the basic keyboard than to type one "special" key.

Because of the desire for touch-typability, it is worth at least considering doing away with such keys as "shift lock." They are rarely, if ever, used and the keyboard space that they occupy is in high demand.

Other things which keyboard manufacturers have done can be deadly. Two examples are illustrative. First, the timing on the shift keys can be blown. The result of doing so is that when "Foo" is desired, "FOo," "fOo," and "foo" are more likely to result. The other example is having a small "sweet spot" on each key. Missing this "sweet spot" will cause both the desired and the adjoining key to fire. Thus, striking "i" can cause "io" to be sent.

More generally, the packaging of a keyboard can be important. Sharp edges near the keyboard or too tightly packed keys can cause errors and fatigue.

5.1.1 Special Function Keys and Other Auxiliary Keys

Keyboard manufacturers seem to have decided that a plethora of special keys is more useful than a more general approach. Thus, you can get "insert line" or "cursor up" or--gasp--"PF1". These keys, when pressed, will either do the function that they name, do something totally random, or send a (usually pre-defined) sequence of characters to the computer. For reasons that have been covered already, having the terminal do the named functions is a losing approach. Having them send pre-defined sequences of characters is not much more useful. For example, the "cursor up" key might send \uparrow E and your editor has this sequence bound to MoveToEndofSentence. Note that this problem exists even though the editor is fully extensible (i.e., it is not an acceptable solution to rebind the \uparrow E command in the editor to MoveUpLine) because the user might still want to use the \uparrow E command for its original purpose. This problem can be avoided if the keys are down loadable with a sequence of characters to send. Thus, the editor can tell the "cursor up" key to send, say, \uparrow P.

Aside from the problems of compatibility with whatever software is being run, the placement of the keys is the worst problem. As has just been stated, keys that are off to one side take too long to hit. Thus, typing is slowed down considerably.

There is yet one more problem. Additional keys are not free and so the number of them that you want to pay for is limited. However, it is desirable to have the ability to specify a large number of functions (i.e., have a large number of codes that can be specified by the user). The number of special keys required grows linearly with the number of codes.

5.1.2 Extra Shift Keys

A more general solution is to provide extra shift keys. These are keys that modify the actions of the other keys. "Shift" and "control" are the two most common examples of such keys. The Teleray 1061 terminal has a "meta" key as an option. This key sets the top (128) bit of the character that is specified. There are thus 256 codes that can be specified instead of the usual 128 from a full ASCII keyboard.

The number of extra shift keys required grows as the log of the number of codes. Thus, 512, 1024, and even 2048 code keyboards are conceivable.

Finding room on the basic keyboard for these extra shift keys is not easy. That is one reason why the removal of the "shift lock" key was suggested earlier. These keys must be on the basic keyboard in order to preserve touch-typability. (It does not take noticeably longer to type the shifted version of a key than the non-shifted version.) The Knight keyboards in use at the MIT Artificial Intelligence Laboratory have several shift keys. They are, unfortunately, located far enough away from the basic keyboard to prevent touch-typability.

5.2 Graphical Input

Another way of interacting with a computer is by means of a graphical input device. The advantage of a graphical input device is that it can reduce the number of commands needed. Such a

device is used for pointing at sections of the screen. It is thus possible to specify items there without having to specify the numerical address of the location or a command string to move you there.

5.2.1 How It Can Be Used

A graphical input device is used by thinking of the screen as one menu with the device pointing to one entry. A cursor of some sort is used to provide feedback about which menu item is currently selected. There are usually one or more "flags" that can be specified conveniently from the device. These flags provide control information. One flag is special and it provides "Z-axis" information.

The basic loop is to track the device with the cursor. When the Z-axis flag is entered, the currently selected action is taken. The screen is logically broken up into two or more sections. One section has the text that is being edited. Moving the cursor here provides a convenient way to move the point around; typing a character could cause it to be inserted wherever the cursor is. Other logical screens can specify menus of possible actions to select from. It is thus a very sophisticated and general way of specifying a position as an argument to a function.

The desired logical screen can be selected by means of the flags or, where the number of flags is limited, by physical position of the cursor on the screen. The Lisp Machine editor and Xerox PARC's Bravo editor both use graphical input devices heavily.

5.2.2 Devices: TSD, Mouse, Tablet, Joystick

There are several types of devices that are either available commercially or experimentally. They shall be discussed in order of usability.

A Touch Sensitive Display (TSD) is just what it sounds like. The screen is covered with a special transparent material that you touch with your finger and it reports the absolute x,y coordinates of where you touched. No "flags" are available until someone can figure out how to track your finger as it brushes the surface as well as when you press more firmly (creating a Z-axis touch). It is the nicest of the devices, although obtaining feedback is hard because your finger covers the most interesting part of the screen.

A mouse is a small box with wheels. It reports the relative movement that you give it (i.e., "he moved me n units up and m units left") as opposed to absolute coordinates ("I am at position x,y"). It can have several flags. It moves along the floor, table, books, legs, or anything else.

A tablet is an absolute version of a mouse (actually, it came first). It can be run with an electronically detected puck (a small box) or a pen. A physical tablet is required for detection and it is usually about 15" x 15" x 1/2". The absolute coordinates are relative to the tablet. There can be several flags for a puck; a pen usually only has Z-axis reporting.

A joystick is a small stick mounted on a couple of potentiometers. It can report either absolute position, first derivative (relative movement) or second derivative. As it is moved small distances, getting good resolution and avoiding "stickiness" and "jumpiness" are hard. It is generally not as nice to use as the others. Flags are usually by means of regular keyboard keys.

Finally, an imaginary but useful device should be considered. That device is a foot-operated

mouse. Using your feet rather than your hand to operate the mouse solves one of most nagging problems of any of these devices, which is that your hands must leave the keyboard with the usual and aforementioned results.

These devices all assume a high bandwidth connection to some computer. Such a connection is not practical over, say, 30 cps phone lines. What must be done in that case is to have the device report to the terminal, which moves the cursor around and reports when a flag has been hit. Thus, it is possible to supply the immediate feedback that is necessary. A 30 cps connection would be quite satisfactory for this operation (but probably not satisfactory for the screen refresh that would follow, say, the selection of a menu).

6. The World Outside of Text Editing

Text editors have been used for many things besides editing text and, in the future, they will undoubtedly be used for more diverse things. Here are some examples:

A text editor can be the primary interface to a mail system. Messages can be composed by editing a buffer and sent with a special command. Mail can be read and managed by reading it into a buffer and having special commands to perform such operations as move to the next message and summarize all messages. Having the full power of a text editor available can make such things as undeleting an accidentally deleted message or copying the text of a message that is being replied to quite easy to implement.

A text editor can be the primary interface to the operating system. Command lines can be edited with the full power of the editor before being evaluated. The past record of interaction can be kept and parts of it examined or re-used in new command lines. If the operating system does not have support for advanced terminals, a display editor can offer its interface for use by other programs. Other programs would then take advantage of the terminal independence of the editor. Alternatively, other programs would insert their output into a buffer and the editor would become an entire terminal management system.

A text editor can be used by a debugger. Multiple buffers and multiple windows can be used to examine (perhaps multiple) source files, interact with the debugger, and see the output/input of the program as it runs. In addition, a debugger might take over an extra window or two to do such things as constantly show selected variables.

A text editor can be an interface to a complicated file. For example, an indexed sequential file can be updated by providing editor commands to read and write entries (adding or deleting them can be managed as well). Within the entry, the full power of the editor is available for editing it.

A text editor can provide a smooth interface to the file system. A directory can be read by the editor and "edited" by the user. Files can be deleted or otherwise changed in a smooth manner by merely moving to the file name and giving a command (e.g., "delete").

(All of the preceding are currently subsystems within Multics Emacs. They are enthusiastically accepted by the user community.)

A text editor can be used to examine and--when absolutely necessary--modify object files. It can thus replace various patching programs.

A text editor is an integral part of a word processing system. Such systems often have features like automatic pagination and continual justification (the document in general and the current paragraph in particular are constantly kept right justified by rejustification after each modification). These features exist in the ALTO editor Bravo, written at Xerox PARC as well as a number of the word processing packages supplied currently for micro computers.

A text editor can deal with proportionately spaced fonts as well as fixed width ones. (The redisplay gets a lot more complex.)

The editor can be interfaced with the compiler to incrementally compile and/or check a program.

Here, the principle of "sticky compiling" must be introduced. Assume that a program has been properly compiled. Now, change a statement by deleting a few characters and inserting a few others. The editor/compiler combination should not give an error message even though the program has been temporarily illegal. Rather, it should be quiet until you have either finished entering the new statement or it is clear that you are making a mistake. (Deciding when you have made a mistake can be hard.) The editor/compiler combination is generally also interfaced with a debugger. This trio supplies the essence of an integrated program development system.

In summary, a text editor can be used for a wide variety of things besides editing text. Taking the intended use into account when designing a new system can provide useful feedback and new constraints on the design of the system as a whole.

I. Annotated Bibliography

This bibliography includes many different types of documents. Some of the documents are user manuals for various editors. Others of them describe the implementation of specific editors. Still others discuss language tradeoffs or input/output system interfaces.

They are grouped by the type of editor that they refer to. Each entry is annotated to help place it in perspective. Documents that are marked with "*" are especially valuable or interesting.

6.1 Emacs Type Editors

There are four principal implementations of Emacs type editors and there are enough documents to justify their separate listing.

6.1.1 ITS EMACS

Ciccarelli, Eugene
 An Introduction to the EMACS Editor
 MIT Artificial Intelligence Laboratory, MIT AI Lab Memo #447,
 Cambridge, Massachusetts
 January 1978

A primer on the editor's user interface.

*Stallman, Richard M.
 EMACS: The Extensible, Customizable, Self-Documenting, Display
 Editor
 MIT Artificial Intelligence Laboratory, AI Memo #519,
 Cambridge, Massachusetts
 June 1979

Provides arguments for the Emacs philosophy.

Stallman, Richard M.
 Structured Editing with a Lisp
 letter in Surveyor's Forum (includes a response by Sanderwall)
 Computing Surveys, vol 10 #4, page 505
 December 1978

This is a response to the Sanderwall paper (referenced later).

On-line Documentation

MIT-AI: .TECO.; TECORD >
 A more detailed command list for TECO
 MIT-AI: .TECO.; TECO PRIMER
 A primer for TECO
 MIT-AI: EMACS; EMACS CHART
 A four page command list for Emacs

MIT-AI: EMACS; EMACS GUIDE
A detailed user interface manual
MIT-AI: EMACS; EMACS ORDER
A more detailed command list for Emacs

6.1.2 Lisp Machine Zwei

*Weinreb, Daniel L. & Moon, David
The Lisp Machine Manual
MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts

January 1979

The user interface for Zwei.

Weinreb, Daniel L.
A Real-Time Display-Oriented Editor for the Lisp Machine
S.B. Thesis, MIT Electrical Engineering and Computer Science
Department, Cambridge, Massachusetts

January 1979

How Zwei works internally.

6.1.3 Multics Emacs

Greenberg, Bernard S.
Emacs Extension Writer's Guide
Order #CJ52, Honeywell Information Systems, Inc.
(In publication)

How to write extensions.

Greenberg, Bernard S.
Emacs Text Editor User's Guide
Order #CH27, Honeywell Information Systems, Inc.
December 1979

The user interface.

*Greenberg, Bernard S.
"Multics Emacs: an Experiment in Computer Interaction"
Proceedings, Fourth Annual Honeywell Software Conference,
Honeywell Information Systems
March 1980

A summary of MEPAP (referenced below).
Also, MIT-AI: BSG; NMEPAP >

Greenberg, Bernard S.
 Read-Time Editing on Multics
 Multics Technical Bulletin #373
 Honeywell Information Systems, Inc., Cambridge, Massachusetts
 April 1978

On-Line Documentation:

(by Greenberg, Bernard S.)

MIT-AI: BSG; LMEPAP >

Why Lisp was chosen for the implementation language

* MIT-AI: BSG; MEPAP >

A detailed history of Emacs in general and the Multics implementation in specific.
 Very valuable.

MIT-AI: BSG; R4V >

A proposal for a terminal independent video terminal support package.

MIT-AI: BSG; TTYWIN >

A look at the good and bad features of video terminals.

6.1.4 MagicSix TVMacs

*Anderson, Owen Ted

The Design and Implementation of a Display-Oriented Editor
 Writing System

S.B. Thesis, MIT Physics Department, Cambridge, Massachusetts
 January 1979

How TVMacs works internally. It concentrates on describing not the editor itself but rather the implementations language: SINE.

Linhart, Jason T.

Dynamic Multi-Window Terminal Management for the MagicSix
 Operating System

S.B. Thesis, MIT Electrical Engineering and Computer Science
 Department, Cambridge, Massachusetts

June 1980

A video terminal management system. Contains many useful comments on terminal independence and redisplay problems.

6.1.5 Other Emacs

This section covers editors which have the same general user interface as an Emacs (e.g., screen-oriented, similar key bindings) but are not extensible or otherwise fall noticeably short of the Emacs philosophy.

Finseth, Craig A.
VINE Primer
Texas Instruments, Inc., Central Research Laboratories, Systems
and Information Sciences Laboratory, Dallas, Texas
August 1979

User interface manual for the complete novice.

Schiller, Jeffrey I.
TORES: The Text ORiented Editing System
revised from S.B. Thesis, MIT Electrical Engineering and
Computer Science Department, Cambridge, Massachusetts
June 1979

On-Line Documentation
CMU-10A: fine.{mss prt}[s200mk50]
User manual for FINE, running at Carnegie-Mellon
University. Written by Mike Kazar.

6.2 Non-Emacs Display Editors

Bilofsky, Walter
The CRT Text Editor NED -- Introduction and Reference Manual
Rand Corporation, R-2176-ARPA
December 1977

Irons, E. T. & Djorup, F. M.
A CRT Editing System
Communications of the ACM, vol. 15 #1, page 16
January 1972

Joy, William
Ex Reference Manual; Version 2.0
Computer Science Division, Dept of Electrical Engineering and
Computer Science, University of California at Berkeley
April 1979

Joy, William
An Introduction to Display Editing with Vi
Computer Science Division, Dept of Electrical Engineering and
Computer Science, University of California at Berkeley
April 1979

Kanerva, Pentti
TVGUID: A User's Guide to TEC/DATAMEDIA TV-Edit
Stanford University, Institute for Mathematical Studies in

the Social Sciences

1973

Kelly, Jeanne

A Guide to NED: A New On-Line Computer Editor

The Rand Corporation, R-2000-ARPA

July 1977

Kernighan, Brian W.

A Tutorial Introduction to the ED Text Editor

Technical Report, Bell Laboratories, Murray Hill, New Jersey

1978

MacLeod, I. A.

Design and Implementation of a Display-Oriented Text Editor

Software Practice and Experience, vol. 7 #6, page 771

November 1977

Weiner, P., et. al.

The Yale Editor "E": A CRT Based Editing System

Yale Computer Science Research Report 19

April 1973

Seybold, Patricia B.

TYMSHARE's AUGMENT -- Heralding a New Era, The Seybold
Report on Word ProcessingVol. 1, No. 9, 16pp, ISSN: 0160-9572, Seybold Publications, Inc.,
Box 644, Media, Pennsylvania 19063

October 1978

On-Line Documentation:

SAIL: E.ALS[UP,DOC]

User manual again. Stanford University.

6.3 Structure Editors

Ackland, Gillian M., et al

UCSD Pascal Version 1.5 (Reference Manual)

Institute for Information Systems, University of
California at San Diego

Donzeau-Gouge, V.; Huet, G.; Kahn, G.; Lang, B.; & Levy, J.J.

A Structure Oriented Program Editor: A First Step Towards
Computer Assisted Programming

Res. Rep. 114, IRIA, Paris

April 1975

Teitelbaum, R. T.
The Cornell Program Synthesizer: A Microcomputer
Implementation of PL/CS
Technical Report TR 79-370, Department of Computer Science,
Cornell University, Ithaca, New York

6.4 Other Editors

Benjamin, Arthur J.
An Extensible Editor for a Small Machine With Disk Storage
Communications of the ACM, vol. 15 #8, page 742
August 1972

Talks about an editor for the IBM 1130 written in Fortran.
Not extensible at all.

Bourne, S. R.
A Design for a Text Editor
Software Practice and Experience, vol 1, page 73
January 1971

User manual

Cecil, Moll & Rinde
TRIX AC: A Set of General Purpose Text Editing Commands
Lawrence Livermore Laboratory UCID 30040
March 1977

Deutsch, L. Peter & Lampson, Butler W.
An On-Line Editor
Communications of the ACM, vol. 10 #12, page 793
December 1967

QED user manual

Fraser, Christopher W.
A Compact, Portable CRT-Based Editor
Software Practice and Experience, vol. 9 #2, page 121
February 1970

Front end to a line editor.

Fraser, Christopher W.
A Generalized Text Editor

Communications of the ACM, vol. 23 #3, page 154
March 1980

Applying text editors to non-text objects.

Hansen, W. J.
Creation of Hierarchic Text with a Computer Display
Ph.D. Thesis, Stanford University
June 1971

Kai, Joyce Moore
A Text Editor Design
Department of Computer Science, Univ of Ill at Urbana-Champaign,
Urbana, Illinois
July 1974

Describes both internals and externals on the editor. However,
the design is a poor one.

Kernighan, Brian W. & Plauser, P. J.
Software Tools
Addison-Wesley, Reading, Massachusetts
1976

This book has a chapter which leads you by the hand in
implementing a simple line editor in RatFor.

*Roberts, Teresa L.
Evaluation of Computer Text Editors
Systems Sciences Laboratory, Xerox PARC
November 1979

A comparative evaluation of four text editors. Quite well done.
Unfortunately, it does not include Emacs (it uses DEC TECO
instead).

Sanderwall, Erik
Programming in the Interactive Environment: the Lisp Experience
Computing Surveys, vol. 10 #1, page 35
March 1978

Talks about the editor for InterLisp.

Sneeringer, James
User-Interface Design for Text Editing: A Case Study
Software Practice and Experience, Vol 8, page 543
1978

User manual and a discussion of user interface concepts.

Teitelman, Warren
InterLisp Reference Manual
Xerox Palo Alto Research Center, Palo Alto, California
October 1978

How to use the InterLisp (non-display) structure editor.

van Dam, Andries & Rice, David E.
On-Line Text Editing: A Survey
Computing Surveys, Vol. 3 #3, p. 93
September 1971

Contains a general introduction to the problems of text editing. Out-dated technology, though.

II. Some Implementations of Emacs Type Editors

This is a partial list and is intended to provide a general guide and not a comprehensive list.

Name	System	Implementation Language
TECO	ITS	Midas (assembler)
Full Emacs		
EMACS	ITS	TECO
Emacs	Multics	Lisp
Emacs	Tops-20	TECO
TVMacs	MagicSix	Sine
Zwei	Lisp Machine	Lisp
Partial Emacs		
FINE	Tops-10	Bliss
MINCE	CP/M	C
otv	MagicSix	PL/1
Tores	UNIX	C
VINE	VAX/VMS	Fortran

III. Partial Emacs Command List

This list is of the command set that is generally common to all of the full Emacses. Specific command bindings can and do vary from implementation to implementation. This list is not complete, nor can it be as commands are constantly being added and changed.

Command designations reflect both the name and the manner in which they are entered. For example, the C-f command is named "control f" and it is entered by typing the ↑F character. Most of the C- commands can be given directly from an ASCII keyboard. Escapes are provided for those that are not. The M-a command is named "meta a" and it is entered from an ASCII keyboard by typing the <esc> key and then the command. Thus, M-a is given by typing <esc> a and M-C-a (or C-M-a) by typing <esc> ↑A.

```

C-@   place the mark at the point
C-a   move to the beginning of the current line
C-b   move backward one character
C-c   a prefix for control-meta commands.  see below
C-d   delete the following character
C-e   move to the end of the current line
C-f   move forward one character
C-g   abort: abort execution of the current command and
      return to the edit loop
C-h   same as C-b
C-i   insert <tab>
C-j   insert <newline>; insert <tab>
C-k   delete the text to the end of the current line; if at
      the end of the line, delete the newline
      character; push deleted text onto the kill buffer
C-l   rebuild the display from scratch
C-m   insert <newline>
C-n   move down one line staying in as nearly the same
      horizontal position as possible
C-o   insert <newline>; move backward one character
C-p   move up one line staying in as nearly
      the same horizontal position as possible
C-q   insert the following character as typed
C-r   search for a string before the point;
      see C-s for details
C-s   search for a string after the point.
      There are lots of things that you can do
      typing characters builds up the search string
      <del> deletes the previous character
      C-s search for the next occurrence of the string
      C-r search for the previous occurrence
      C-g abort
      <alt> terminate search; if the string is null, the
      previous string is used
C-t   interchange the characters on each side of the point,
      leaving the point after the second one; if at
      the end of a line, interchange the previous
      two characters
C-u   universal argument.

```

There are two forms

C-u C-u <command> do <command> 4, 16, 64, 256, ...
 times depending upon the number of C-us.
 C-u <integer> <command> do <command> <integer> times.
 (e.g., C-u 3 5 C-f means to C-f 35 times)

C-v move the bottom of the current screen to the top of the
 screen

C-w delete the text between the point and the mark; push
 the deleted text onto the kill buffer

C-x a prefix for control-x commands. see below

C-y copy the top item from the kill buffer to the point;
 place the mark at the beginning of the
 block and the point at the end

C-z return to superior

C-[a prefix for meta commands. see below

C-\ a prefix for meta commands. see below

C-]

C-† a prefix for control commands. See this list

C- _

!"#\$%&'()*+,-./ insert themselves
 0123456789
 ;:<=>?@
 A..Z
 []†'_
 a..z
 { | } ~

bs,back space
 same as C-h

tab same as C-i

lf,line feed
 same as C-j

cr,carriage return,return
 same as C-m

esc,escape
 same as C-[(the <alt> key)

del,delete,rubout
 delete the previous character

C-<alt> you are now typing at whatever is running the editor

C-% ask for the old string, then the new one and replace
 all occurrences of the old with the new

C-/ give help

C-< place mark at the beginning of the buffer

C-> place mark at the end of the buffer

C-? give help

C-x C-b print a list of all buffers and associated information

C-x C-d display the current directory

C-x C-f ask for the name of a file and read it into a buffer
 whose name is derived from the filename; if

there is a conflict with an existing
buffer, you are asked for a name to use

C-x C-i indent the region
C-x C-l convert the region to lower case
C-x C-o delete the blank lines around the
point
C-x C-p move to the top of the current screen; place the mark
at the end of the current screen
C-x C-r ask for the name of a file and read it into the
current buffer
C-x C-s write out current buffer to the current filename if it
has been modified
C-x C-u convert the region to upper case
C-x C-w ask for the name of a file and write the buffer to
that file
C-x C-x exchange point and mark
C-x 1 use one window
C-x 2 use two windows
C-x 3 use two windows and stay in the first
C-x = print where you are in the buffer
C-x A ask for the name of a buffer; append the region to that
buffer
C-x B ask for the name of a buffer and put you there
C-x D edit directory
C-x F set the fill column to the horizontal position
C-x I run INFO
C-x M send mail
C-x O in two window mode go to other window
C-x R read mail
C-x ↑ grow window by
C-x a same as C-x A
C-x b same as C-x B
C-x d same as C-x D
C-x f same as C-x F
C-x i same as C-x I
C-x m same as C-x M
C-x o same as C-x O
C-x r same as C-x R

M-~ ditto. copy the word directly above onto this line
M-% QueryReplace. ask for an old string and a new string
At each occurrence of the old string, it
is displayed and you are asked for a command
<sp> replace and go on
 don't replace and go on
, replace and wait
. replace and exit
<alt> exit
↑ return to previous old string (jump to mark)
C-w delete old string and enter C-r recursively
C-r normal edit, but recursively invoked
C-l redisplay screen
! do not ask any more

M-(insert "()"; leave point between them
M-< move to the the beginning of the current buffer
M-> move to the end of the current buffer
M-? help
M-A move to the beginning of the current sentence
M-B move backward one word
M-C capitalize the following word
M-D delete the following word; push deleted text onto
the kill buffer
M-E move to the end of the current sentence
M-F move forward one word
M-G fill text in the region
M-H move to the beginning of the current paragraph;
place the mark at the end of the
current paragraph
M-L convert the following word to lower case
M-Q fill the current paragraph (make each line as long
as possible); C-u M-Q means do justify
(same, but make right margin even)
M-S center the current line on the screen
M-T interchange the adjoining words, leaving the point
after the right hand word
M-U convert the following word to upper case
M-V move the top of the current screen to the bottom of
the screen
M-W push a copy of the region onto the kill buffer
M-X ExecuteCommand
M-Y (after C-Y) delete yanked text and yank previous
kill buffer entry
M-[move to the beginning of the current paragraph
M-\ delete the <sp> and <tab>s around the point
M-] move to the end of the current paragraph
M-a same as M-A
M-b same as M-B
M-c same as M-C
M-d same as M-D
M-e same as M-E
M-f same as M-F
M-g same as M-G
M-h same as M-H
M-l same as M-L
M-q same as M-Q
M-s same as M-S
M-t same as M-T
M-u same as M-U
M-v same as M-V
M-w same as M-W
M-x same as M-X
M-y same as M-Y
M- delete the previous word; push deleted text onto the
kill buffer
C-M-) move up one level of list structure backward

C-M-(move up one level of list structure forward
C-M-A move to the beginning of the current defun
C-M-B move backward one S-expression
C-M-E move to the end of the current defun
C-M-F move forward one S-expression
C-M-G format the current S-expression
C-M-H move to the beginning of the current S-expression;
place the mark at its end
C-M-K delete the following S-expression; push the
deleted text onto the kill buffer
C-M-O move the rest of this line vertically down,
inserting <tab>s and <sp>s as needed
C-M-T interchange the adjoining S-expressions, leaving the
point after the following S-expression
C-M-W the following delete-and-push will be part of the
current entry in the kill buffer
C-M- delete the preceding S-expression; push
the deleted text onto the kill buffer

