HONEYWELL

MULTICS EMACS TEXT EDITOR USER'S GUIDE

SOFTWARE

MULTICS

EMACS TEXT EDITOR USER'S GUIDE

SUBJECT

Tutorial Introduction to the Emacs Text Editor, Full Description of the Editing Requests Available, and Instructions for Using Special Features of Emacs

SPECIAL INSTRUCTIONS

This manual presupposes some basic knowledge of the Multics system provided by the two-volume set, *New Users' Introduction to Multics*. Some of the preliminary information covered in that set is summarized briefly here, however, so that users at any level of experience can comprehend the techniques presented in this manual.

SOFTWARE SUPPORTED

Multics Software Release 10.2

Includes update pages issued as Addendum A in July 1981, Addendum B in January 1982, Addendum C in August 1982, Addendum D in February 1983, and Addendum E in December 1983.

ORDER NUMBER

CH27-00

December 1979

Honeywell

PREFACE

This book is a detailed description of the Multics Emacs text editor, a real-time editing and formatting system designed for use on video terminals. It is intended for all users; both those who have relatively little experience on the Multics operating system (or any other computer system) and experienced programmers will find this a complete description. Users are, however, expected to be familiar with the Multics concepts described in the two-volume set, New Users' Introduction to Multics--Part I (Order No. CH24), and--Part II (Order No. CH25), referred to in this manual as New Users' Introduction .

Although Multics Emacs is easily used by technically inexperienced people, those with some programming experience can utilize it even more effectively by writing their own extensions. Examples of supplied extensions are the Emacs message subsystem and the various language modes, described in Appendices B and C of this manual. Information about extensions and instructions for writing them are provided in the Emacs Extension Writers' Guide, Order No. CJ52, which is referred to in this book as Extension Writers' Guide. The Introduction to Emacs, Order No. CP31, is recommended for users interested in a simpler introductory manual that describes a basic subset of Emacs editor requests.

Many video terminal types are supported by Emacs, as supplied. Information on how to support additional terminal types is also available in the Extension Writers' Guide. It is recommended that at least one person at your site have a copy; someone should also be familiar with the Emacs installation information.

The information and specifications in this document are subject to change without notice. This document contains information about Honeywell products or services that may not be available outside the United States. Consult your Honeywell Marketing Representative.
©Honeywell Information Systems Inc., 1983 File No.: 1L33, 1U33 CH27-00

The term "file" is used interchangeably with "segment" in this manual, since many of the editing requests have the word "file" as part of their command names. Emacs requests operate both on single-segment and multisegment files. Zeros are slashed in this manual (Ø) in a few instances when they might be confused with the letter O.

The sections of this manual fully describe the Emacs editor and explain the steps required to edit effectively any type of user text. In general, the basic techniques are explained first; more powerful or efficient requests are introduced as you proceed through this book. The first sixteen sections are tutorial; Section 17 summarizes, by editing function, all the Fundamental mode requests, including both those described in the tutorial and some additional requests that are used less frequently.

Section 1 is a brief introduction.

Section 2 describes how to begin: using the terminal, logging into the Multics system, and entering the Multics environment.

Section 3 tells how to enter text, move the cursor, make simple corrections, and log out.

Section 4 describes a few requests for deleting text and retrieving deleted text.

Section 5 explains how to read and write files (segments).

Section 6 describes some simple search requests for locating character strings.

Section 7 introduces requests for manipulating blocks of texts.

Section 8 describes numeric arguments and various other ways to reexecute and reverse editing requests.

Section 9 defines an Emacs "word" and describes the requests that operate on words.

Section 10 deals with screens and buffers, telling how to display different areas of the buffer on your screen, how to switch buffers, list them, and delete them.

Section 11 includes the help facilities available on Emacs. The editor is completely documented online, so information is always available during editing.

Section 12 defines Emacs sentences and paragraphs, and describes requests that operate on them.

Section 13 includes many requests for handling white space, indentation, and formatting.

Section 14 gives more information on manipulating blocks of text, inserting files, and using named regions and marks.

Section 15 describes keyboard macros that can easily be created to perform special editing tasks.

Section 16 describes the use of multiple windows, the window editor, and the buffer editor.

Section 17 contains descriptions of all the Fundamental mode requests, arranged by function.

Appendix A documents the emacs command and lists, alphabetically, all the Fundamental mode requests.

Appendix B describes the Emacs mail mode for sending and reading electronic mail.

Appendix C describes the Emacs programming language modes and their requests, which are tailored for use in writing and editing programs in Lisp, FORTRAN, PL/I, and ALM.

Appendix D describes the Macro Edit mode for editing keyboard macros.

Appendix E gives instructions for using Emacs on printing and glass teletype terminals.

Appendix F describes the Emacs message facility for accepting and responding to interactive messages.

Appendix G details how to write an Emacs start-up to customize the environment automatically each time Emacs is entered.

Appendix H describes pop-up-windows mode, which dynamically creates and removes windows as they are needed.

Appendix I describes the list_emacs_ctls command, which lists all known Emacs terminal types.

Appendix J describes the overwrite-mode minor mode, which alters the ways characters are inserted into the buffer.

Multics Emacs was modelled after the EMACS editor at the MIT Artificial Intelligence Lab. EMACS (on the AI Lab PDP-10's) was written, in TECO, by staff members of the MIT AI Lab and the (MIT) Laboratory for Computer Science (LCS), without whose encouragement and support this project would not have been possible.

Significant Changes in CH27-00F

Emacs now supports multisegment files (MSFs). The much larger capacity MSFs can be read and written, using the same Emacs commands used to read and write single-segment files. Thus, multisegment file is assumed throughout this manual, where file and segment are used interchangeably.

Emacs affords greater protection against inadvertent destruction of file and buffer modifications by querying whether to proceed with write-file, save-file, read-file, and find-file requests, when file or buffer contents have changed since last read or written.

In compliance with above, new options have been added to the opt command (see find-file-check-dtcm, save-same-file-check-dtcm, read-file-force, and write-file-overwrite). Also, defaults have changed for the find-file-set-modes, remember-empty-response, and paragraph-definition-type options.

The default compiler option for pl1-compile-option is now the null string.

This page intentionally left blank.

CONTENTS

CONTENTS (cont)

ILLUSTRATIONS

SECTION 1

INTRODUCTION

Multics Emacs is an integrated editing, text preparation, and screen management system designed to take advantage of the features of modern display terminals. Text entry and editing on these video screen display terminals are done interactively. You, the user, can see the effects of Emacs editing on the screen as you type.

This manual is arranged so that you, as a new Emacs user, can learn Emacs by immediately beginning to use it. The first part, Sections 2 through 16, are tutorial in nature, and cover text entry and the more basic Emacs requests. Section 17 summarizes the requests covered in the preceding sections, and introduces the remaining Fundamental mode (basic Emacs mode) requests. Advanced users should immediately turn to this section, which provides short descriptions of every Fundamental mode request. The requests are presented there in functional groups, i.e., for a particular type of editing task, all the requests available to perform that task are described.

Users who work through the first sixteen sections will also find Section 17 useful for reference and for learning the additional requests not covered in the tutorial.

The appendices describe specialized uses of Emacs, including the Emacs mail system and programming language modes. Appendix A provides the emacs command description, and an alphabetized list of the Fundamental mode requests.

Throughout this manual, "Emacs" designates the system, and the all lowercase "emacs" designates the Multics command invoked to use the system.

SECTION 2

GETTING STARTED

THE TERMINAL

Although it can be used on printing terminals, Emacs has been designed especially for use on screen terminals. Sit down at your screen terminal and note the three parts you will be using as you edit:

- the screen
- the keyboard
- the modem communicating between the terminal and Multics

Figure 2-1 shows a typical screen terminal and Figure 2-2 shows a typical keyboard and the special keys described below.

The Screen

The screen of your terminal is like a television screen, and displays the information needed to communicate with Multics and Emacs. Messages from the system appear on the screen, and your responses, typed on the keyboard, also appear.

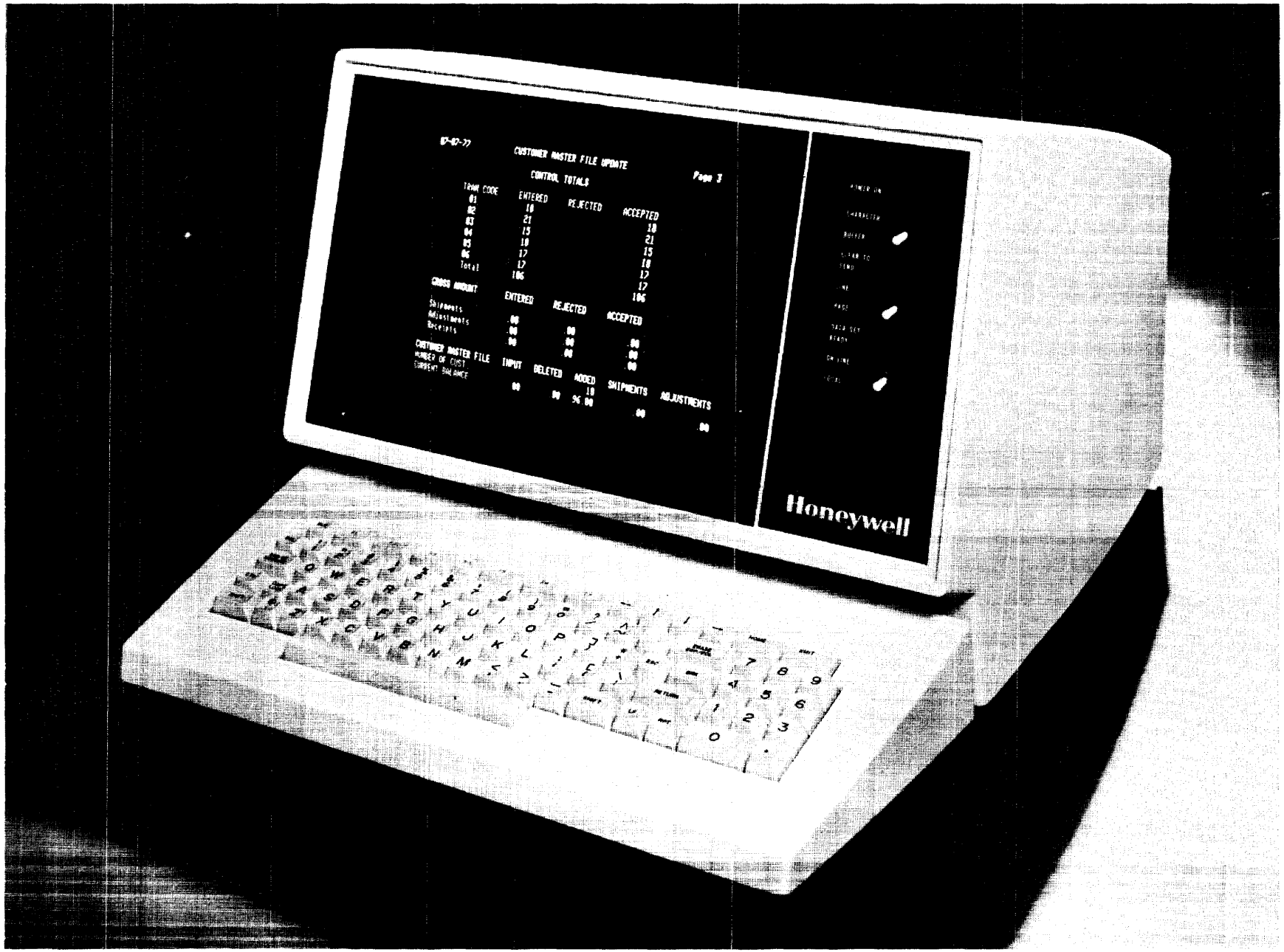


Figure 2-1. A Screen Terminal

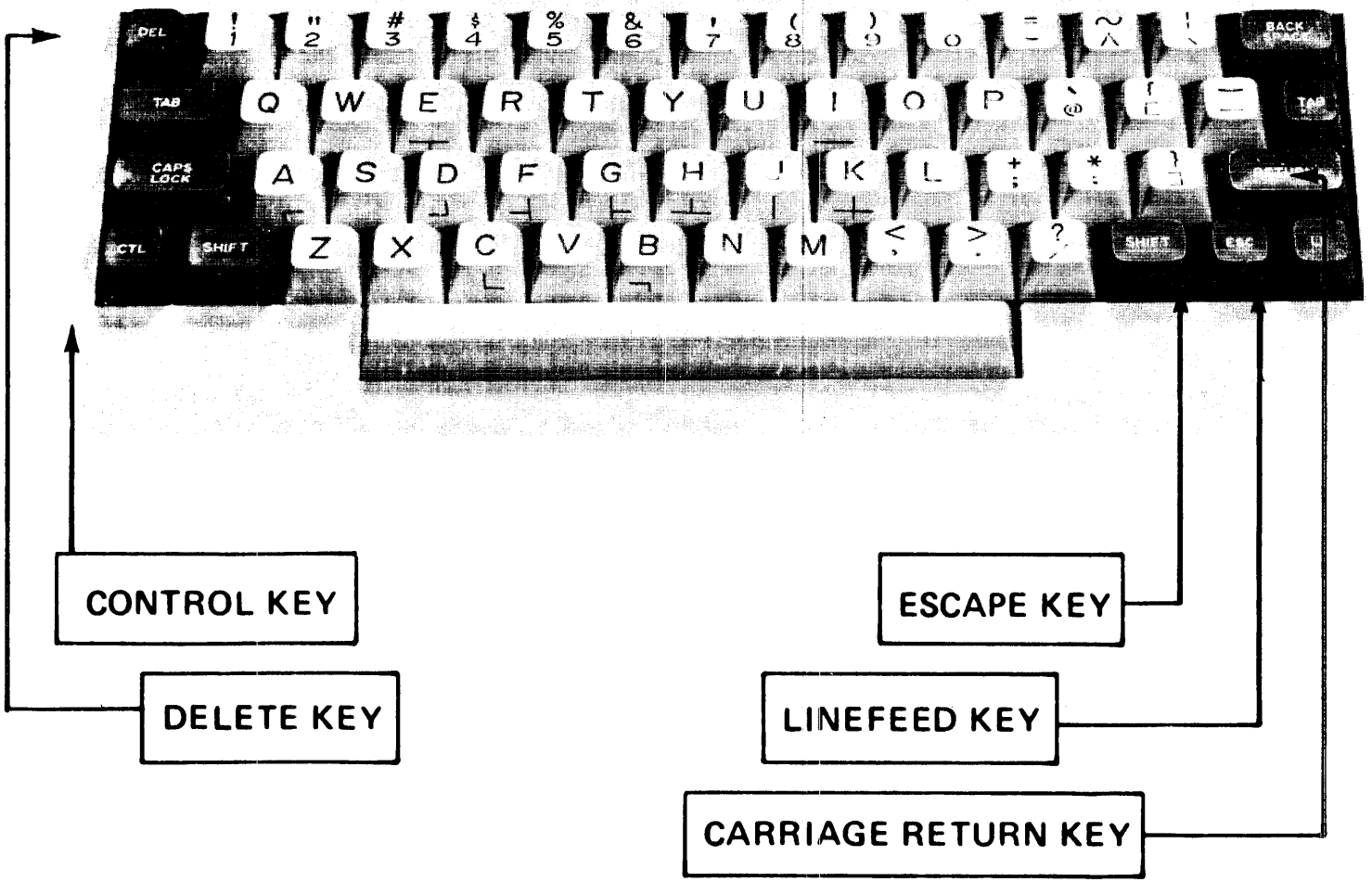


Figure 2-2. A Terminal Keyboard

The Keyboard

Your keyboard resembles the keyboard of a typewriter, with its letters and special characters, but has additional keys. Several of them are important for Emacs usage. They include the following:

- control key
- escape key
- linefeed key
- delete key
- carriage return key

CONTROL KEY

Terminals vary, but you should be able to locate a key labelled with the letters CTL, CTRL, CONTROL, CNTRL, or something similar. This is the control key. It operates like a shift key in that it must be held down while you hit one or more normal characters. Simply pressing it and releasing it has no effect. For example, if you press the "p" key, you get a lowercase p. If you press the "p" key while holding down the shift key, you get an uppercase P. If you press the "p" key while holding down the control key, you get something called a control P. This control P is a control character. All control characters are interpreted as requests to Emacs. Control characters are used to control Emacs, to manipulate the cursor and text. In this manual, the ^ symbol represents the control key; alphabetic characters following the ^ symbol are represented as uppercase, even though, for them, a control character is the same whether the shift key is held down or not (and generally you would not hold it down).

ESCAPE KEY

The next key you should locate is the escape key. The escape key is commonly labelled with the letters ESC, ESCAPE, ALT, or ALTMODE. On some terminals, you may have to hold down the shift key to get an escape. Unlike the control key, which is held down while a normal character is typed, the escape key is typed sequentially, i.e., before or after a normal character. You use the escape key for some Emacs requests and to terminate your response to a few of the Emacs prompts. Be sure to release the ESC key quickly, to avoid getting two (or more) escapes in a row. In this manual, the letters ESC represent the escape key.

LINEFEED KEY

Your keyboard should have a linefeed key labelled with the letters LINEFEED, LF, or NEW LINE. The linefeed key is sometimes used in Emacs.

DELETE KEY

The delete key transmits the ASCII DEL character, octal code 177. The key is generally labelled with the letters DEL or RUBOUT. As its name suggests, its use is to rubout, or erase, the previously typed character(s). In this manual, the character sequence \177 represent the delete key.

CARRIAGE RETURN KEY

On Multics, a carriage return returns you to the left margin and inserts a newline character in your input. This key is often labelled either RETURN or CR.

The Modem

Your terminal must be connected to the Multics system in some manner for emacs, or any other Multics command, to work. The modem or an acoustic coupler provides an interface to the communications link between your terminal and Multics. Many modems are equipped with a telephone receiver and dial. For this type of modem, you dial a specific number to begin the logging in procedure and make the connection to Multics. However, a wide variety of modems exist; if you do not know how to establish the connection between your terminal and Multics, you should ask a technically qualified person at your site to help you log in.

Technical Requirements

Two technical requirements that your terminal must meet are that it be an ASCII terminal, and that it be capable of running in full duplex mode (i.e., have controllable local echo). Either your terminal or your modem may have a switch that can be positioned to half or full duplex mode; you should set this switch to full duplex. If your terminal does not have controllable local echo (printer on/off), you should log in in full duplex and echoplex modes. Generally, your site will have arranged for appropriate terminal modes to be set automatically when you log in. If you find characters are printed out twice, setting the modes and/or switches should correct the problem. Again, if you have a question about either of these requirements, ask someone for help.

If your terminal has an auto-linefeed key or switch, be sure it is off, and use lfecho mode to achieve its effect. Failure to do this results in certain displays vanishing from the screen prematurely.

LOGGING IN

The first thing you want to do is establish a connection with the computer. This is called logging in. To log in, you must be registered on the system, as a member of a certain project. You are given a unique User_id, user identification, which consists of a Person_id (name) and Project_id (project name). For example, Mary Smith, working in the sales department, may be given the following User_id:

```
Smith.Sales
```

This User_id belongs to Mary alone; no one else can use it. Mary also has a password, which along with her User_id allows her to use the system.

The procedure for logging in is explained in depth in the New Users' Introduction - Part I. Briefly, however, to log in you turn power on for the terminal, dial the appropriate telephone number, and when you hear a beep signal, either press a button or place the telephone receiver in the modem and wait. (This method is employed unless your terminal is directly connected to Multics, in which case you do not need to dial a phone number.) When a connection has been established, a header of the following type is displayed by Multics on the terminal:

```
Multics 8.0: PCO, Phoenix, Az.
Load = 26.0 out of 100.0 units: users = 26
```

At this point, type the login command and your Person_id, separated by a blank, and then a carriage return. For example:

```
login Smith
Password:
```

Multics then requests your password (the second line, above). Depending on your terminal, the display of the password is either suppressed or hidden in a string of cover-up characters typed by the system. Video terminals usually just suppress tha password. If you make an error while logging in, the system informs you of it and asks you to try again.

```
Login incorrect.
Please try again or type "help" for instructions.
login Smith
Password:
```

After you have successfully typed your password, the system responds with information regarding your last login.

```
Smith Sales logged in 06/07/80  0937.5 mst Tue from ...
Last login 06/06/80  1359.8 mst Mon from terminal...
```

The last line of system-generated text in the log-in sequence is the ready message. This message is printed to indicate that Multics is at command level and ready to receive the next command. The ready message consists of the letter "r" followed by the time of day and two numbers that reflect system resource usage. For more information about the ready message, refer to the ready command in the MPM Commands.

```
r 12:22 3.229 1799
```

The complete log-in sequence for Mary Smith is:

```
login Smith
Password:

Smith Sales logged in 06/07/80  0937.5 mst Tue from ...
Last login 06/06/80  1359.8 mst Mon from terminal...
r 12:22 3.229 1799
```

INVOKING THE emacs COMMAND

You have logged in and received the ready message indicating that you are at command level. To enter Emacs, type the emacs command on your keyboard, followed by a carriage return (Multics command lines are always terminated by a carriage return):

```
emacs
```

Emacs then checks your terminal type against the list of known terminal types supported by emacs, which are known as Emacs video terminal controllers (CTLs). If it finds a controller for your terminal, Emacs uses that controller. If your terminal type is not found, it checks the current terminal modes to see if your terminal is a video display terminal (CRT) or a hardcopy printing terminal. If your terminal is a printing terminal, Emacs displays the file the best that it can, given the limited capabilities of printing terminals. If you have a video terminal of an unknown type, Emacs tells you:

```
Unknown terminal type.

Do you want a list of known terminal types?
```

If you type "yes", Emacs displays the names of all terminals it knows how to support. (This list is the same as the one generated by the list_emacs_ctls command, described in Appendix I.) Among these names, you should be able to find the acceptable form of the name of your terminal, or the name of a supported terminal that most closely matches the characteristics of your terminal. After displaying the list, Emacs asks:

```
What type terminal do you have?
```

Type it in, followed by a carriage return. If you cannot find the name of an acceptable terminal type, type:

```
quit
```

followed by a carriage return, to return to command level. You may want to try again on a different terminal type. (Instructions for supporting new terminal types are available in Emacs Extension Writers' Guide, Order No. CJ52, "Writing Emacs Terminal Control Modules.")

The Initial Display

Once Emacs has recognized your terminal type, either automatically or by querying you as described above, it takes several seconds to get started. When it has started up, it clears the screen, and displays the line:

```
Emacs 11.11 (Fundamental) - main
```

at the lower left of the screen. This line is called the mode line. The mode line tells you several things, the most important of which is that you are communicating with Multics Emacs, rather than with the command processor or with another editor. A number may follow the word "Emacs"; it gives the Emacs version in use. The name of the major mode you are in is parenthesized, and here in the above example it is the simplest major mode, Fundamental mode. Emacs has several modes best suited for different tasks, such as preparing text or programs in the various programming languages. Major modes each have a distinct set of key bindings (typed key sequences that specify particular request instructions to Emacs). Minor modes provide "fine tuning" to modify the way the Emacs works, but do not have a special set of key bindings. The names of minor modes, if you are using any, appear right after the major mode name in the mode line, enclosed in angle brackets (<>):

```
Emacs 11.11 (Fundamental <overwrite>) - main
```

You can edit several different items at once with Emacs. Each item being edited is edited in a separate editing environment called a buffer. The buffers are named so that you can identify them. The buffer name in the mode line above is "main".

The area between the top of the screen and the mode line is where text being edited appears. This area of the screen is called the window. The window always displays about twenty consecutive lines of the text you are editing. Of course, if your text is shorter than that, part of the window is empty.

At this point, the only sign of life in your window is a blinking object in the upper left corner. This is the cursor. It may be a blinking underline, or a blinking or solid box, depending on your terminal. The cursor is the most important thing to watch on your screen. It is always on some position on the screen, and all "action" occurs at the cursor. All the text you enter is entered at the cursor (and the cursor moves), and all the text you delete is deleted at the cursor.

SUMMARY OF TERMS

When the emacs command is invoked, the first screen displayed is pretty simple, since it is practically empty. You should, however, be familiar with the following terms, and be able to relate them to what appears on the screen.

- mode line
- major mode
- key bindings
- minor mode
- buffer
- buffer name
- window
- cursor

SECTION 3

ENTERING TEXT AND SIMPLE CURSOR MOVEMENTS

TYPING IN TEXT

Now you are going to enter some text. To do so, you simply begin typing. Type the following on the terminal, and stop after typing the period:

This is sample text.

In general, you may have to wait a few moments for the characters you are typing to appear on the screen, especially if it has been a few minutes since the last time you typed. If the user load on Multics is especially heavy, the wait is correspondingly longer.

The text appears at the top of the screen as you type. This line is the beginning of your document; the way it looks is exactly the way your document is. Stop and look at the screen. The cursor is to the right of the period in the sentence just typed. You should get into the habit of thinking of yourself as being "at" some point in your document. That point is indicated on the screen by the cursor.

At the bottom of the screen, an asterisk (*) appears in the line below the mode line. This indicates that the buffer has been modified, i.e., changed in some way. It remains until the modified buffer is written out, and reappears thereafter whenever new modifications occur that have not been written out. This is discussed more in Section 5.

Now add another line Hit the carriage return key (for a Multics newline) This puts a carriage return into your text and moves the cursor to the beginning of the next line Type:

```
Here is more text yet.
```

Again, the text appears on the screen as you type it. The first line you typed stays where it is, with the second line appearing under it. Your document is now two lines long, and you could continue to type in more text indefinitely. It is just like using a typewriter.

EDITING WITH EMACS REQUESTS

Emacs performs all of its editing functions by carrying out programmed instructions when you issue a request. You issue, or invoke, a request by typing certain keys or key sequences. Each request (key or key sequence) is individually attached, or bound, to a command name, which tells Emacs what set of instructions to follow when you issue the request associated with it. A command name is a hyphenated, abbreviated (sometimes) name that describes the action of the request to you, and specifies the appropriate instructions to Emacs. As you learn Emacs, command names serve to remind you of what the requests do. When you are proficient in Emacs, however, the command names can be used in computer programs to construct your own requests. As a matter of fact, you can "connect" (bind) any key of your choice to any command name of your choice if you do not like the default requests provided. The section on keyboard macros describes how to do so.

Self-Inserting Characters

When you type in new text, you are actually issuing Emacs requests. Printing characters (other than #, @. and \, whose special meanings are explained later) are called self-inserting because when you type one, it inserts itself into the text in your buffer. So typing the letter "d", for example, tells Emacs:

```
What:   to insert a "d"
Where:  at the cursor
```

Correcting Typing Errors with Emacs

Even the best typist makes an occasional error. The following are special characters useful for making corrections with a single keystroke:

- # (The erase character)
- \177 (the delete key)
- @ (the kill character)

To see how these characters work, you will have to type a line with an error in it. Go to the next line, by typing a carriage return, and type:

```
Multix
```

Stop as soon as you type the "x".

THE ERASE CHARACTER (#)

On Multics, if you type a wrong letter, you use the # (pound or number sign) character to erase it. The same is true in Emacs; the command name of the # request is rubout-char. Type the # character and watch what happens. The "x" in Multix disappears from the screen, leaving no trace of itself or the erase character. The cursor, which was positioned right after the x, backs up a space and is now positioned right after "i" in Multi. No trace or record of the mistake remains. You could, at this point, simply type in the letters c and s, and your correction would be completed. However, type an "x" back in, so that you can see another way to correct typing errors.

THE DELETE KEY (\177)

The screen should be set up now exactly the same as it was when you tried the erase character a minute ago. With the cursor right after the "x" in Multix, hit the delete key once. Again, the x disappears and the cursor moves back one space to appear right after the "i" in Multi.

Typing either a # or the delete key erases the previous character, i.e., the character right before the cursor. Use whichever is most convenient. The command name of the delete key request, as you might expect, is also rubout-char.

THE KILL CHARACTER (@)

Suppose that you decide you would like to erase the whole line, which now consists of the letters "Multi." The cursor is still positioned directly to the right of the trailing i. On Multics, you use the @ (commercial-at sign) character to erase everything typed so far on the current line. In Emacs, the @ also erases everything on the current line to the left of the cursor, and the remaining text moves to the beginning of the line. Try typing it. You see the five letters displayed disappear, and the cursor move to where the "M" was. The @ request's command name is kill-to-beginning-of-line.

Typing in Special Characters

If you are wondering how to actually insert one of the special characters, like the #, into your experimental text, here is an additional thing to try out.

^Q

The ^Q request, whose command name is quote-char, "quotes" the character immediately following it, i.e., it tells Emacs to insert it literally into the buffer. For example, if you hold the control key down and type a q, then release the control key and type a pound sign (holding down the shift key on your terminal if necessary), a # appears on your third line. No characters are deleted. Now type a # without preceding it by a ^Q, and the # in your text disappears.

MOVING THE CURSOR

You could continue to add text to your two-line document by just typing it, ending each line with a carriage return, and simply typing at the end of the document to add more lines (but always ending that last with a carriage return), and correcting any mistakes you catch with the #, \177, and @. However, you would probably soon wish to change or modify some text already entered. In order to do so, you need to know how to move the cursor to the place requiring the change. The following paragraphs demonstrate how to move the cursor:

- to the previous line (^P)
- to the next line (^N)
- forward a character (^F)
- backward a character (^B)
- to the beginning of the line (^A)
- to the end of the line (^E)

Getting to the Right Line

In order to change the line you entered that reads:

"This is sample text"

to

"This is some sample text"

you first have to get from the beginning of the empty third line on your screen to the first line, and then to the right place in that first line. To do this, you move the cursor up one line at a time by telling Emacs to go to the previous line.

^P

The Emacs request that moves the cursor to the previous line is ^P (control P). The command name is prev-line-command. Locate the control key, hold it down while you press the p key, and then release both keys. Now watch the screen: you see the cursor move up to the previous line, to a place right above where it had been. Note also that when you type the ^P, you do not "see" it appear on the screen, but only its effect. What you see on the screen is what you have in your document, and it does not matter what you used to achieve that state.

You still need to go up one more line, so type another ^P.

Moving Within the Line

^F

Now that you are on the line in which the change is to be made, you need to be able to move the cursor forward to the correct point on that line. The Emacs request for moving forward a character at a time is ^F, forward-char. Hold down the control key and slowly press the f key a few times. The cursor moves to the right, one character at a time, as many times as you type ^F. Continue to type ^Fs until the cursor is under (or covering, on some terminals) the letter s of the word "sample". This is where you plan to add the word "some".

^B

If you go too far to the right, you have to "back up" to the s in "sample". The Emacs request for moving backward a character at a time is ^B, backward-char. So, regardless of whether you did in fact go too far or not, try moving the cursor backward. Again, hold down the control key and hit the b key several times. The cursor moves backward to the left one position for each ^B you type. Soon you reach the beginning of the line. You cannot go any further back, since this is the beginning of your text, so Emacs causes your terminal to beep, indicating that you have made an error. Simply wait for the beeping to stop, and release the control key.

Now, go ahead and move the cursor forward again to the s in "sample". To add the word "some", just type it in. Emacs moves the rest of the line over and displays your text:

This is somesample text.

Obviously, this is not quite the way you want this sentence to appear. To add the space between the two words, simply hit the space bar. Now you see:

This is some sample text.

with the cursor still under the s in "sample". The text is fixed.

Note that you did not have to do anything special to type in the new word. You just moved the cursor to the right place and started typing.

Getting Back to the Right Line

^N

In order to add a few more lines to the end of your document, you must move the cursor down a couple of lines. You do this by going to the next line, and then the next line after that. As you may have guessed, the Emacs request for going to the next line is ^N, next-line-command. Hold down the control key and type an n. The cursor moves down to the next line, but note its position in that line. Emacs tries to keep you in the same column when going between lines, so the cursor is under the initial t in the word "text". When you hit the n key again, still holding down the control key, the cursor moves down another line but goes all the way over to the left. This line is empty. Although Emacs tries to keep you in the same column, it chooses the column in a reasonable fashion. In general, you would not consider it useful to be positioned in the middle of an empty line. Likewise, if the word "Multix" had still been on this line, Emacs would have placed the cursor immediately to the right of the x, the closest column on this much shorter line.

The Ends of the Line

^A

To try out the next two line-movement requests, type in:

using Emacs is easy

To say instead, "I think using Emacs is easy," you could type a string of ^Bs to move backwards to add the first two words. However, an easier way exists (using Emacs really is easy). Emacs provides the ^A request (go-to-beginning-of-line) to move the cursor to the beginning of the current line. Type a ^A and watch your screen. The cursor moves to the u in "using". You can now type:

I think

remembering to end with a space so that the "somesample" problem does not recur.

^E

The only thing now missing from your sentence is punctuation. Rather than typing a series of ^Fs to move forward a character at a time, you can skip right to the end of the current line with a ^E, go-to-end-of-line. The ^E request positions the cursor right after the last character, and before the carriage return if the line has one. On a line with nothing on it (nothing in it but a carriage return), ^E does nothing. Try the ^E, watch the screen, and then type either a period or an exclamation point, depending on your state of enthusiasm!

GETTING STOPPED

You may wish to practice and experiment with the Emacs requests you have just learned. Feel free to do so, since none of this text will be saved (reading in and writing out files is covered in Section 5). When you are finished you can exit from Emacs and log out of Multics.

Exiting from the Editor

When you began this session, you logged into the Multics system. You were then at command level, and, therefore, able to invoke the Multics command, emacs, to enter Emacs. When you want to leave Emacs, you must return to command level by issuing an Emacs request. Once returned to command level, you can invoke the Multics logout command, and end the session. Figure 3-1 illustrates this process; an imaginary "you are here" arrow would point to the lower Emacs box.

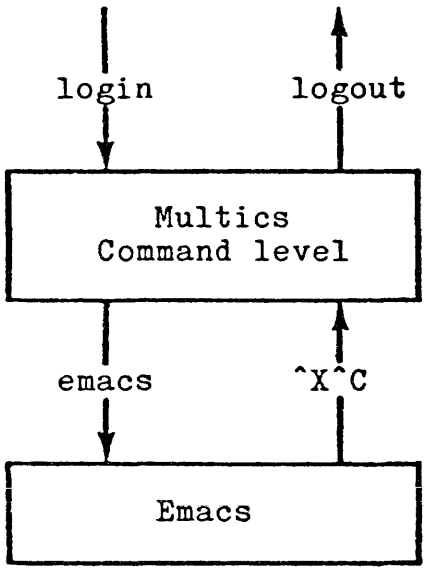


Figure 3-1. Editor Entry and Exit

^X^C

The Emacs request that returns you to command level is ^X^C, quit-the-editor. The control key must be held down while you depress first the x and then the c keys. Try it. As you watch the screen, the top several lines of your text is replaced by the following message from Emacs:

```
Modified Buffers:
> * main
----------------
```

At the same time, Emacs writes a message and question, called a prompt, in the area below your mode line at the bottom of the screen. The question is called a prompt because Emacs is "prompting" you for a response. This area where the prompt appears is called a minibuffer, and normally occupies two lines (though both are not always used). Emacs prints this message in the minibuffer:

Modified buffers exist. Quit?

Since your buffer has changed, from being empty to having text in it it is now modified. Emacs is telling you that you have done work that will not be saved, and making sure that you really want to quit under these circumstances. If you had saved your work by writing it out to a file, Emacs would not prompt you at all, but simply return you immediately to command level.

However, since you are not going to save this practice work, simply type your response:

yes

and a carriage return. Prompts require not only a response, but also a termination character, generally a carriage return, to indicate the response's end. When you type the carriage return, angle brackets (<>) appear in the minibuffer at the end of your response.

In the case of the prompt above, when you type "yes" and a carriage return, Emacs clears the screen and returns you to Multics command level. A ready message appears at the top of your screen to indicate that you are, indeed, at command level.

If you mistype your response, Emacs indicates that you've typed an inappropriate response to its query. At the top of the screen, the line that said "Modified Buffers:" will say instead:

Please answer "yes" or "no"

This not only lets you know you have given the wrong response, but also provides you with the acceptable choices. You can answer either "yes" or "no" (or use short forms "y" for "yes" and "n" for "no"). If you catch the error before typing a carriage return to end the prompt, you can edit your response just as you would edit text.

SUMMARY OF TERMS

A few new terms have been introduced that you should remember, since they are going to be mentioned frequently.

- request
- command name
- prompt
- minibuffer
- modified buffer

Logging Out

From command level, you can now invoke the Multics command for logging out, logout. Logging out breaks the connection between your terminal and Multics. After you have typed it, the system responds by displaying your identification, the date and time of the log out, and the total CPU time and memory units used.

```
logout
Smith Sales logged out 06/07/80  1249.4 mst Tue
CPU usage 17 sec, memory usage 103.1 units.
hangup
```

The word "hangup" is displayed by Multics to remind you to hang up the telephone and to indicate that the connection has been broken on purpose.

SECTION 4

SIMPLE DELETING AND KILLING

DELETING CHARACTERS

You already know how to use the # to delete mistyped characters. This is especially useful if you catch an error immediately after typing it, since it erases the single character to the left of the cursor. To learn another method, log in again and invoke the emacs command. You can now type in the following text, exactly as it appears here so that you can follow this lesson easily (do not type a carriage return after "computerized."):

> There's no denying that computer technology has made our lives easier. Computers serve us at home and at our jobs. In publications wirk, three areas made much faster by automation include text entry, editing, and formatting. Someday even typesetting will be computerized.

The cursor should now be to the right of the period following "computerized". In reviewing this text, you see that in the third line, "wirk" should actually say "work". To correct this in Emacs, first you have to "go" to that point, by moving the cursor. Type two ^Ps, the previous line request, and your cursor moves to after the "h" in "much". The backward character request, ^B, gets you to the letter "i" in the offending "wirk". Since you have to move backward several characters, you should hold down the control key and hit the b a dozen or so times. If you go too far, use ^F to go forward to the correct spot. The cursor should end up under (or covering) the "i" like this (the cursor is represented as an underscore):

our jobs. In publications w_irk, three areas made much

Deleting One Character at a Time

^D

You now want to change the i at the cursor to an o. In Emacs, this change is made by deleting the unwanted character(s), and typing in the desired one(s). You delete characters, one character at a time, by typing ^D, delete-char. Type one ^D and watch the screen. The third line now appears as:

our jobs. In publications wrk, three areas made much

The "i" has been removed from the line, the text has closed up, and the cursor is under the "r". Now type the correct letter, and the line reads:

our jobs. In publications work, three areas made much

The cursor is again under "r" that follows the "o" that you added.

Deleting Lines

^K

Often, you need to delete a lot more than a few characters; you want to remove entire lines, large pieces of lines, or many lines. The kill lines request, ^K (kill-lines), does this. For example, typesetting already is computerized, so you might want to replace that sentence with something else. Again, you must reposition the cursor and then correct the text.

First, get to the letter "S" in "Someday" by using the backward character, forward character, next line, and previous line (^F, ^B, ^N, ^P) requests. When the cursor is under the "S" type a ^K, and watch the screen. All the text between the cursor and the end of the line vanishes.

RETRIEVING KILLED LINES

You could replace the line just killed by typing in a new sentence right here and now, but suppose you decide instead that you really want it back. (The terms "delete" and "kill" both mean to erase text, but generally delete applies to characters deleted one at a time, and kill applies to characters deleted as groups, i.e., lines and sentences.) You can get it back intact because it has been preserved in a special place called the kill ring.

The Kill Ring

The ^K request puts killed text in the kill ring, and it stays there so that you can retrieve it, either right away or many requests later. Often it is useful to purposely kill text so that you can move it from one place to another by killing it at one position in your text, repositioning the cursor, and then retrieving it from the kill ring. Whether you purposely or accidentally kill text, however, the kill ring provides the security of being able to conveniently recover it.

The kill ring normally has ten "slots" for saving your text. When you issue a kill request (you know only two so far, @ and ^K), the killed text goes into the first slot. (The # and ^D requests do not put the characters they delete into the kill ring. If you accidentally delete a character, it is easy enough to retype it; if you delete many characters by hand, it is probably not a mistake and thus not important to be able to recover them.) If you later issue another kill request, the text previously killed moves into the second slot, and the newly killed text goes into the first slot just vacated. Killed text keeps rotating down a slot in this fashion until all ten slots are filled. At the next kill request, the very first killed text would be discarded, since no eleventh slot is available in which to save it.

Kill merging is a feature that allows you to save related killed text in the same slot in the kill ring. For example, if you are looking at several lines, and decide to kill them, you probably start at the top and delete them line by line. Since you delete them as one, they should be stored as one, so that they can be brought back as one. Kill merging provides this; if you type successive kill requests that kill text in the same direction, the text killed by each request is merged and occupies only one slot.

So, for kill merging, the kill requests must be:

- successive kill requests
- in the same direction

Successive kill requests have no intervening keystrokes between them; you cannot type in any new text or issue any non-kill Emacs requests. If you do, the killed text goes into separate slots. These successive kill requests must also kill text in the same direction in order to merge, i.e., the requests must both/all eliminate text from either right to left (forward), or left to right (backward). The ^K kills forward. The @ kills backward (however, note that you cannot do successive @ kills, so they would never merge anyway).

SUMMARY OF TERMS

Before moving on to see how killed text actually is retrieved, you should be sure to understand the terms dealing with where it is retrieved from:

- kill ring
- kill merging
- successive kills

Yanking Text Back

While you have been reading the above, the "Someday even typesetting..." sentence has been sitting in the kill ring. Maybe you want to reconsider using it, and would like another look.

^Y

You can "yank" it back out of the kill ring by typing a ^Y. The ^Y request, named yank, is useful for letting you fix damage done by mistaken or inadvertent killing, and for moving lines around. Type a ^Y and watch the screen. The sentence is back, and the cursor is positioned after it.

Reposition the cursor, by means of the requests you know, to the "S" in "Someday". Remove the sentence again with ^K and type in:

```
Speed is essential, since technological
advances must be documented to be used.
```

The correction is made, and this new sentence begins where the killed one began.

MORE ABOUT ^K

The ^K request does different things depending on whether it is used at the end of a line or not. Type about four ^Ns to get to a fresh place on the screen to enter some new text. Note that the cursor is at the left margin; these lines are empty. Type the following well-known verse:

```
I wandered lonely as a cloud
That floats on high o'er vales and hills,
When all at once I saw a crowd,
A host, of golden daffodils;
Beside the lake, beneath the trees,
Fluttering and dancing in the breeze.
```

Position the cursor to under the "I" starting the verse. Now kill all the text on that line with a ^K. Observe the screen: the line becomes blank. Now type ^K again (a successive forward kill), and watch the screen. All the rest of the poem moves up. To summarize the actions of ^K:

- When anywhere but at the end of a line (i.e., at the beginning or in the middle of a line), ^K deletes all text between the cursor and the end of the line, leaving the cursor at what is now the end of the line.

- When at the end of a line, ^K removes the carriage return, or "sticks the next line onto the end of this one," making one line. If the line that the cursor is on has nothing in it (except the carriage return), this makes the line go away. If the line that the cursor is on contains more than a carriage return, the next line is tacked onto the end, as though you never hit a carriage return between them.

Now type another ^K. The line:

That floats on high o'er vales and hills,

empties out. Type it once more, and that (now empty) line disappears, and the cursor is at the beginning of the next line of the poem. Typing successive ^Ks deletes lines one by one as you type them.

To yank these lines back, type one ^Y. Both lines come back because the text was merged in the first slot on the kill ring. Try typing another ^Y, and watch the screen. If you were expecting to get the "Someday even typesetting..." sentence yanked back, you will be surprised to see the two poetry lines be duplicated on your screen. Just because they have been yanked

back once does not mean they are no longer in the kill ring. In fact, they still occupy the first slot, and you can yank them back into your text anywhere as many times as you want, as long as they remain in the kill ring. The ^Y yanks text from the first slot. When text is pushed into succeeding slots by subsequent kills, it is retrieved by giving a numeric argument to the ^Y request (essentially, providing the number of the slot from which text should be yanked). Numeric arguments are explained in Section 8.

One additional note: sometimes more text gets yanked back than you actually want. However, going back and killing any unwanted text that you have yanked is much easier than having to retype killed text that you could not yank.

SECTION 5

WRITING AND READING FILES

This section explains how to save your edited text by writing it out to a file, and how to read existing files into Emacs for editing (files and the Multics storage system are explained in detail in the New Users' Introduction - Part I).

WRITING A FILE OUT

Before you write the contents of the "main" buffer out to a file, put about twenty more lines into it. Just type in additional short lines of any text you choose. End the last line, as all lines, with a carriage return (if you do not, your file will not end in a newline character, and many Multics programs will not operate properly on that file). The object is to have enough text to experiment with while learning the remaining Emacs requests. Be sure, however, to embed these lines somewhere in what you add:

```
Now is the time, and the only time,
for those who have the time to
give their time.
```

^X^W

The Emacs request for writing a file is ^X^W, write-file. When you finish adding text, type ^X^W and watch the screen. Emacs prompts in the minibuffer:

```
Write File:
```

The cursor has jumped from the last character you typed into the minibuffer, and is waiting for you to supply the pathname of the place you want to write the buffer's contents. Type in a pathname, and end the prompt with the carriage return. For example:

```
first.practice
```

This creates a segment named first.practice in your working directory. As Emacs writes this file, the word "Writing..." appears in the minibuffer. When the word "Written." replaces "Writing...," you know that the file has been successfully written out to the Multics storage system. At the same time, the full pathname of the file appears right below the mode line. This is called the path line, and tells you exactly what file you are working with. (You are, in fact, only working with a copy of the file; any additional changes you make would not be reflected in first.practice until you write it out again.) You may have noted the asterisk (*) that began the path line. It appears there whenever a modified buffer has not yet been written out. Hence, when you issued the ^X^W, it disappeared. It reappears with subsequent modifications, remaining until the next writing out of the buffer. It is a convenient indicator for determining whether or not you need to write out the buffer in order to save your work.

The bottom of your screen now looks something line this:

```
Emacs (Fundamental) - main
   >udd>Sales>Smith>first.practice
     Write File: first.practice<>
     Written.
```

You have the mode line and the path line, followed by the two lines in the minibuffer that provide extra "status" information.

If you already have a file named first.practice in your directory, Emacs verifies the write-file request with the query

first.practice already exists. Overwrite it?

This gives you the opportunity to reconsider before overwriting the file contents, by specifying yes or no to the query in the minibuffer.

You can disable this protection feature either by preceding write-file with a numeric argument (e.g., ^U^X^W) or by turning on the write-file-overwrite option (it is off by default) as described under "Additional Optional Settings" in Section 17.

Is Your New File Really There?

To verify that you have indeed written out a segment called first.practice, leave Emacs and return to command level. The ^X^C request does this. Now you can invoke the Multics list command, which lists the segments (in your working directory in this case). (The list command is described in detail in the New Users' Introduction - Part I.) Type:

```
list
```

The segment named first.practice should be the first file listed.

READING A FILE IN

When your files have been listed, and you have a ready message, reenter Emacs by invoking the emacs command:

```
emacs
```

You can now read in a file.

When you read in a file, it is read into a buffer; every file read in goes into its own buffer. The first twenty or so lines appear on the screen. Although you see only these first several lines, the whole file is there. If you try to position the cursor to an unseen line, either by doing ^Ns so that the cursor tries to go off the bottom, or ^Ps so that the cursor tries to go off the top, Emacs displays the lines so that the one you want is indeed shown.

^X^F

The find file request for reading in files is ^X^F, find-file. Type it and watch the screen. The cursor drops below the mode line, into the minibuffer, which reads:

```
Find File:
```

with the cursor after the colon. Emacs is prompting you for the pathname of the file to be read in.

Type the pathname of the file that you previously wrote out, ending with a carriage return. In our example, you would type either:

```
>udd>Sales>Smith>first.practice
```

or, if you are in the directory that contains first.practice, simply:

```
first.practice
```

If you make a mistake while typing in the minibuffer, you can edit it with #, ^B, ^D, ^F, or any other request, before you type the carriage return. Assuming you have typed it correctly, Emacs prints "Reading..." in the minibuffer, and strikes it out when it has finished reading. The screen fills up with the first windowful of the file. The cursor is at the first character of the first line.

Now look at the mode line. The buffer is no longer "main," but "first." The buffer name is taken from the first component of the entry name of the file read in (in this case, the file was first.practice, so the buffer is named "first"). The path line, below the mode line, gives the full pathname of the file in this buffer.

If the buffer already contains first.practice and the file being read in has been changed since the buffer was last read or saved (perhaps by another user or program), Emacs responds with a local display at the top of the screen

```
Buffer first contains an old version of first.practice
```

A query in the minibuffer offers several choices, including a help option to explain the possible actions

```
Select "overwrite", "use", "skip", "new" buffer, or "help":
```

Respond with help and you are told

```
Since buffer first was last saved or read, the file
     first.practice has been modified.
     The buffer HAS NOT been modified since then.

     Respond with one of:
     overwrite - to reread the file into this buffer
     use       - to use this buffer as is
     new       - to select a new buffer
     skip      - to skip the current file
```

The cursor is then placed back at the prompt so that you can make your selection.

You may first be queried on which buffer to use before having to select an action, if for example, the default buffer is already in use or several buffers contain the same file.

If you have made modifications to the buffer, the local display will say

Modified buffer first contains an old version of
first.practice

and the help message will contain the words HAS ALSO instead of HAS NOT.

You can bypass this check and just read in the file by turning off the find-file-check-dtcm option (it is on by default) as described under "Additional Optional Settings" in Section 17.

A more convenient way to read in this file is to supply the file's pathname as an argument to the emacs command. To do so, simply type the emacs command, a space, and then the pathname:

emacs first.practice

Thus, instead of invoking the emacs command, starting in the main buffer, and then reading first.practice into the buffer "first", you enter Emacs and read first.practice directly into the buffer "first" in one step, bypassing the "main" buffer.

Counting the Lines in a File

Once a file is read in, you may want to know how big it is.

^X=

The line counter request tells you how many lines are in your document, the number of the line in which the cursor is currently positioned, and the cursor's dprint column position. This request is ^X=, linecounter. Additionally, if your file does not end in a newline character, ^X= informs you.

Type a ^X and an equal sign (=). Be sure to release the control key before striking the equal sign. Also, remember to hold the shift key down if the equal sign requires a shift on your terminal.

Emacs prints something similar to this in the minibuffer:

38 lines, current = 1, column = 1

This sample document is 38 lines long, and the cursor is on the first line in the first column.

Now type ^Ns slowly until the cursor is on the line right above the minibuffer. Be careful, just this time, to type slowly enough so that the cursor stays in the window currently displayed. Now note the contents of the line it is on. Then strike one more ^N. Emacs rewrites many lines on your screen, and, if your terminal has the capability to do so, moves many lines around. When it is finished, the cursor is on a line in the middle of your screen. Look at the line above that one. You see that it is the same line that was at the bottom of the screen a minute ago.

Type another ^X=. The minibuffer now says something like this:

38 lines, current = 22, column = 1

If you look at the screen, the cursor is on about the twelfth line, not the 22nd. However, you are editing the file (or the buffer), not the screen. The line you are on is, in fact, the 22nd line of the buffer. Since only about 21 lines can be displayed at once, Emacs automatically chooses which 21 lines to display in order to make sure that the line you want is on the screen.

You can now edit this file as though you had typed it in, using all the requests you know. Try putting in a new word or killing a line. At once, the word:

Modified

appears in the minibuffer, letting you know that you have changed the file since you read or wrote it, and have to write it out if you want your work to be saved.

One service ^X= performs is to warn you if your file does not end in a newline (carriage return). If it does not, because you forgot to end the last line in it with a carriage return, ^X= prints a message like this:

38 lines (NO NEWLINE), current = 23, column = 15

In such a case, you should go to the end of the last line and insert a carriage return.

SAVING (REWRITING) A FILE

Having made some editing changes to the file, you must write it out to save them.

^X^S

The request that writes out the same file that you read in is ^X^S, save-same-file. This request uses the default pathname of the file when writing out the buffer's contents. This means that when you type a ^X^S, Emacs recognizes that you wish to use a pathname that it already knows. Essentially, you are telling Emacs, "Since no pathname is supplied here, you should, by default, use one already supplied." The default pathname is always the pathname that appears in the path line. The ^X^F request sets the default pathname, and typing ^X^S causes Emacs to "go get" that pathname and write the buffer out to the segment named therein.

Try typing ^X^S. You notice that Emacs does not prompt for a pathname, but does repeat the "Writing.../Written." message in the minibuffer to let you know when it is done.

If Emacs discovers that the file being saved has changed since the buffer was last read or written, it queries whether to proceed with the save-same-file request

<pathname> has changed since last read or written. Save anyway?

This allows you to reconsider before overwriting the contents of the file (answer yes or no as appropriate).

You can disable this protection feature either by preceding save-same-file with a numeric argument (e.g., ^U^X^S) or by turning off the save-same-file-check-dtcm option (it is on by default) as described under "Additional Optional Settings" in Section 17.

ADDITIONAL NOTES ON WRITING FILES

Access Restrictions

You may encounter a problem sometimes when you try to write out a file that you have read in without any trouble. This occurs if you have read (r) access to the file, but do not have write (w) access (access requirements are discussed in the New Users' Introduction--Part I). You will be notified that you have an access problem by an error message like this:

Incorrect access on entry.

If Emacs displays such a message, you cannot use ^X^S; you must write out the file with ^X^W and supply a different pathname from the one used to read in the file.

The Default Pathname with ^X^W

When writing out a file with ^X^W, Emacs prompts you for a pathname. If you wish to write the file out to the default pathname set by ^X^F, simply type a carriage return in response to the prompt. The ^X^S request performs the same action, and is, of course, slightly more convenient.

If Emacs discovers that the file has been modified since it was last read into the buffer, it will query whether to proceed with write-file to the default pathname

<pathname> has changed since last read or written. Save anyway?

You can disable this protection feature either by preceding write-file with a numeric argument (e.g., ^U^X^W) or by turning off the save-same-file-check-dtcm option (it is on by default) as described under "Additional Optional Settings" in Section 17.

SUMMARY OF TERMS

Two new terms introduced above that you need to remember are:

- path line
- default pathname

SECTION 6

LOCATING A SEQUENCE OF CHARACTERS

A fundamental ability needed in editing is that of looking for a particular sequence of characters, or searching. In Emacs, this means finding a given sequence of characters in the buffer, and moving the cursor to that point.

To try out these next requests, you should start at the beginning of your file. Use ^Ps to get there.

SEARCHING FORWARD

When you search forward, Emacs starts where the cursor is and searches toward the end of the buffer.

^S

The string search request is ^S, string-search. To locate the first occurrence of the word "time," type a ^S. Emacs responds in the minibuffer:

String Search:

The response to this prompt is to type in the string to search for, in this case, the letters t, i m, and e. Then type a carriage return. All prompts in Multics Emacs, including those for search strings, end with CR. When you type the CR, you see the cursor return to the main window, but it is now immediately after the e in the first occurrence of the word "time."

Try looking now for the word "grime." Type a ^S, type the letters g, r, i, m, and e, and hit the CR key. Emacs responds:

Search fails.

in the minibuffer, since no "grime" was found between the cursor's position at the end of "time" and the end of the buffer. The cursor remains where it was.

Try locating "time" again as you did before. The cursor advances to the right of the next "time" in the buffer. The ^S only searches forward, never backward. It puts the cursor after the end of the string it finds so that Emacs will not keep finding the same one.

You can take a shortcut in locating the third "time." Type a ^S, but when Emacs prompts for the string, simply hit the CR key. Emacs puts the word "time" in the minibuffer just as though you had typed it. When you answer a search request's prompt with an "empty" search string, i.e., a CR only, Emacs reuses the last search string (sequence of characters) you were searching for. This use of a default search string applies to all the search requests. This way, you can search for the same thing many times without retyping it into the minibuffer. Search for "time" a couple more times, just so you will be positioned after the embedded sentence.

A Word About Search Strings

The string for which you search does not have to be a complete word; it can be any number of characters. Emacs searches for the string exactly as typed, so you can include whitespace characters in your search string to define it more narrowly. Since a CR ends the prompt, however, you must use the quote-char request, ^Q, before any CR that you wish to include in your search string.

GETTING OUT OF TROUBLE

^G

An important Emacs request lets you "get out" of what you are doing. This is the ^G, command-quit. Try typing it; your terminal beeps. The ^G does more than just beep, however. Type another ^S, and Emacs again prompts:

String Search:

Suppose you decide that you did not want to search for anything, or you typed ^S in error. Simply type ^G; the cursor exits from the minibuffer, the terminal beeps, and the search request is aborted. This request can always be used to exit the minibuffer to abort any prompting request, such as ^X^F or ^X^W, that you change your mind about in midstream. If you ever find yourself in the minibuffer with a prompt that you do not understand, or think you did not ask for, typing ^G will get you out without doing any harm.

The fact that ^G causes a beep can also be used to let you know when Emacs has "caught up" to you after a large number of cursor movement requests. When you type a ^G at the end, you know that the beep means that Emacs is responding to the last request, so the others are done. This is sometimes useful when the system is slow.

^X^G, ^Z^G, AND ESC ^G

The ^X^G, ^Z^G, and ESC ^G requests are similar to ^G. When you type a prefix character, i.e., ^X, ^Z, or an ESC, by mistake, these requests undo it, i.e., they flush the prefix character. So, if you type ^X, ^Z, or ESC in error, just go on to type ^G right after them to get out of it. All three requests also cause the terminal to beep. However, unlike ^G, they do not exit the minibuffer. These requests are all named ignore-prefix.

SEARCHING BACKWARD

^R

To search in a backward, or reverse direction, type ^R (reverse-string-search). Emacs prompts in the minibuffer:

Reverse String Search:

Search for the word "time" again; since that was the same string used in your last search request, you need type only a CR in response to the prompt. The string "time" appears in the minibuffer, and the cursor is left before (i.e., under the first character of) the first occurrence of "time" going backward from where the cursor was when you typed ^R. If the cursor was right after a "time," it is now right at the front of the same one.

Repeat the process. The cursor goes one "time" back each time, until there are no more between the cursor and the beginning of the buffer. Emacs then responds:

Search fails.

in the minibuffer.

GENERAL RULES FOR SEARCHING

The ^S and ^R requests both:

- prompt in the minibuffer
- take a string terminated by CR to use for a search target (a CR within the search string must be preceded by the ^Q request)
- use an empty string (just a CR) to indicate that the last search string used should be used again

The reverse string search (^R) goes:

- backward to the beginning of the buffer
- leaves the cursor before the located string

The string search (^S) goes:

- forward from the cursor to the end of the buffer
- leaves the cursor after the located string

When searching for something in Emacs, you generally know if you want to search forward or backward for it. If you do not know, search forward. If the search fails, just type a ^R and a CR, and Emacs searches backward for the same string. If the search fails again, the string is not in the buffer.

LOCATING AND REPLACING STRINGS AUTOMATICALLY

ESC %

A powerful request that allows you to search forward for a specified string, and replace that string with another, is ESC %, query-replace. After you type and release the escape key, type the % character, and you are prompted for the string to search for and the replacement string. Type both in the minibuffer (they are individually prompted for), and end both prompts with CR. This request locates, by searching forward, the first string, positions the cursor immediately after the string, and waits for one of the following responses (type the appropriate keys):

space
: replaces this occurrence of the first string with the second. Then searches for the next occurrence of the first string, updates the screen and waits for a response again.

CR
: leaves this occurrence of the first string unchanged and searches for the next occurrence of the first string, again waiting for a response after locating it.

. (period)
: replaces this occurrence of the first string with the second and then terminates the query replace.

^G
: leaves this occurrence of the first string unchanged and terminates the query replace.

ESC
: same as ^G

!
: replaces all occurrences of the first string from the current point to the end, without querying again.

, (comma)
: replaces this occurrence of the first string with the second, immediately updating the screen. Then searches for the next occurrence of the first string and waits for a response again.

^L
: redisplays the screen.

^_
: displays a description of the allowable responses (i.e., prints this list).

?
: same as ^_.

This request allows you to substitute one string for another selectively throughout your buffer. Try replacing some of the occurrences of "time" with "grime."

SECTION 7

WORKING WITH BLOCKS OF TEXT

MARKING A REGION

Often you wish to delete an arbitrary extent of text, i.e., from "here to there," without the tedium of carefully killing individual lines or characters. This extent, or block of text, is called a region in Emacs. In order to delete a whole region, you first must be able to define its limits, or boundaries.

Setting the Mark

One limit of the region to be deleted is determined by the position of the point at the time the region-deleting request is given. The cursor, on most terminals, is under or over a character. Its left edge, however, can be described as being always BETWEEN two characters, the character at the cursor and the character preceding that one. The point is this position between the characters, indicated by the left edge of the cursor.

A further distinction between the cursor and the point is rather fine. However, the point is the theoretical location of the cursor's left edge at a specific time, or where it would be if it had gotten there by the time Emacs takes action as though it had already. The cursor itself is often a moving object; its actual location at a specific time may be somewhere between where it was and where it was going to be when Emacs interrupts its journey to send it somewhere else. Emacs knows where it was going to be, however, and that is the point! Figure 7-1 graphically illustrates the difference between the cursor and the point.

First Stage	Second Stage	Third Stage
Cursor and point between Ds.	User types ^P^P^P^D. Point between A and A. Cursor on way up.	Cursor and point at top line between A and the newline.

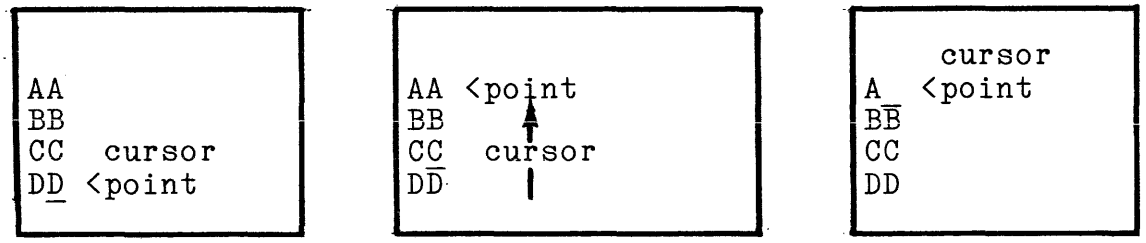


Figure 7-1. The Cursor and The Point

So, one of the region's limits is the point. The other limit is specified by an "invisible cursor" called the mark. Each buffer has only one current mark, and has none at all until you set it. However, whenever you reset the mark (i.e., set it in a new position), the old value of the mark is saved on a mark ring that works like the kill ring. Section 8 describes how to retrieve these "saved" marks. You cannot delete a region until both limits define its boundaries.

^@

To set the mark, you first must move the cursor to where you want the mark to be. You are going to delete this portion of the Wordsworth poem:

> Beside the lake, beneath the trees,
> Fluttering and

Move the cursor to under the B in "Beside." Set the mark there (i.e., at the point in front of this "B") by typing ^@, (set-or-pop-the-mark). This may be tricky if your terminal requires a shift to get the commercial at sign, since you will have to hold down both the control and shift key while you hit @. On some terminals, e.g., Digital Equipment Corporation terminals, you have to type the space bar while holding down the control key to send the ^@ character. The word "Set" appears in the minibuffer to let you know the mark is set.

Now move the cursor to the space between "and" and "dancing" in the next line, i.e., right after the last letter of the last word to be deleted. This is the point, and your region is defined by the two limits.

Exchanging the Mark and the Point

Before deleting a region, you often want to verify that you do indeed know where the mark is. By exchanging them, i.e., switching the cursor and the mark's positions, you can do this at a glance.

^X^X

The ^X^X request, exchange-point-and-mark, makes this exchange. Try it. The cursor suddenly appears under the B in "Beside." Type another ^X^X and everything is as before.

DELETING A REGION

^W

You are now ready to "wipe" out, or delete, the marked region. Type ^W (wipe-region). All the text between the mark and the point disappears from the screen. The newline character within the region also is gone, so you end up with:

```
A host of golden daffodils;
 dancing in the breeze.
```

You still have a space before "dancing;" this is because the space was not included in the region to be deleted.

Yanking a Region Back

The ^W request, like the ^K request, puts whatever it removes into the kill ring. Type a ^Y and watch the text reappear. A useful feature of the ^Y request is that it automatically sets the mark (or resets it if it was previously set) at the beginning of the text it retrieves, and leaves the cursor after it. This has two implications that should be noted:

- After yanking text with ^Y, for any reason, you can delete it again simply by typing ^W, since the cursor and the mark exactly specify the limits of what was yanked.

- After yanking text with ^Y, for any reason, you can move the cursor to the beginning of the yanked text by exchanging the point and the mark with ^X^X.

Your cursor, now in the middle of the last line of poetry, can go to the B in "Beside" if you type ^X^X. Try it. Now, even though the cursor is at the beginning, and the mark at the end of this region, try typing ^W. The region disappears. As long as a region has both limits set, ^W works whether the point or the mark is the first limit.

ESC Y

Sometimes you kill some text, move the cursor, kill more text, and then decide that killing that first text was a mistake. What you want to do in this case is return to where you killed that first text and yank it back. So you move the cursor back and type ^Y. Unfortunately, that retrieves the text of the second kill; the most recently deleted text occupies the first slot on the kill ring. So now you want to get rid of the retrieved text and retrieve the text of the previous kill.

Well, ESC Y, wipe-this-and-yank-previous, does this. It deletes the text just yanked, and rotates the kill ring so that text from the previous kills moves up a slot, and then it retrieves the text now in the first slot. Try a couple more ^Ws, and then ESC Y (hit and release the escape key before typing a y). By doing many ESC Ys in a row, until you "find the kill that you want," you can "go shopping" in the kill ring for saved text.

SUMMARY OF TERMS

Many Emacs requests besides ^W use regions. You should, therefore, understand the following terms:

- region
- point
- mark

SECTION 8

REPEATING AND UNDOING REQUESTS

Emacs provides a variety of ways for repeating the action of a request, either once or many times. Requests for which it is meaningful and useful to specify "how many times" to do them generally accept a numeric argument. A numeric argument is essentially a repetition count, a number that means "repeat this action this number of times." Positive numeric argument generally repeat the request's action in a straightforward manner. Negative numeric arguments generally reverse the action of the request and repeat it; i.e., they cause a request to act like its complement, if it has one.

NUMERIC ARGUMENTS

If you want to say, "Go five characters forward," or, "Go four next lines down," you can give the ^F and ^N request positive numeric arguments of 5 and 4, respectively.

You give a numeric argument by typing the escape key, ESC, then typing the number you want, and, finally, typing the request. For example, to delete six characters, you would type;

ESC 6 ^D

(spaces are unnecessary and incorrect, but have been included for legibility.) That is, strike and release the escape key, type a 6, and then type a ^D. All the characters disappear at once. Try it.

Similarly, if you type:

ESC 249 ^N

while on the first line of a large file, the screen would fill immediately with lines 240 to 260 (approximately) of the file, with line 250 and the cursor in the middle of the screen (you can verify the line number with ^X=). You do not have to watch the cursor step through 250 lines one by one. Experiment with your file and the ^N and ^P requests. Deliberately give a numeric argument too large for your file, and observe what happens.

A negative numeric argument is given by typing ESC -N, where N is the number, before typing the request. With negative arguments, ^F goes backward the specified number of characters, ^N goes to the Nth previous line, ^D acts like # or \177 and deletes N characters to the left of the cursor, etc. An argument of ESC -1 or, simply, ESC-, just reverses the action and performs it once. Some requests do not accept negative arguments; you receive a message in the minibuffer stating this if you give them one by mistake.

Requests Accepting Numeric Arguments

Of the requests already learned, the following accept numeric arguments.

^F	Forward Character
^B	Backward Character
^N	Next Line
^P	Previous Line
#	Rubout Character
\177	Rubout Character
^D	Delete Character
^K	Kill Lines
^Y	Yank
^@	Set/Pop the mark
^S	String Search
^R	Reverse String Search

The first seven, when given either a positive or negative numeric argument, perform the action that number of times. The last five act in a special manner. ^K, ^Y and ^@ do not accept negative arguments. ^S and ^R accept either positive or negative arguments.

When you give ^K a positive numeric argument, it kills that many entire lines, starting at the current point on the current line. Everything killed is put, as one, on the kill ring, and will kill merge with preceding or following killed text as described earlier. Whereas ESC 4 ^K kills four lines, four ^Ks would not, since you generally must type two ^Ks to kill one line of text.

When you give ^Y a positive numeric argument, it yanks text out of the slot on the kill ring that corresponds to the number given. As you have seen, a simple ^Y request retrieves the latest thing killed, from the first slot. If you want, for example, the second latest thing killed (from the second slot), simply give the ^Y request a numeric argument of 2 by typing:

ESC 2 ^Y

When you give ^@ any positive numeric argument, it "pops" the previous mark off the slot on the mark ring and sets the current mark at the position where that saved mark was originally set. If text was deleted from around that position, the current mark is set at the position closest to its former position.

When you give ^S a positive numeric argument, for example 5, it searches forward for the 5th occurrence of the string. If there aren't 5 occurrences, it leaves the point immediately after the last occurrence, gives you a message saying how many occurrences it found, and sets the mark at the original location.

When you give ^R a positive numeric argument, for example 5, it searches backward for the 5th occurrence of the string. If there aren't 5 occurrences, it leaves the point in front of the last occurrence, gives you a message saying how many occurrences it found, and sets the mark at the original location.

When you give ^S a negative numeric argument, for example -5, it searches backward for the 5th occurrence of the string. In other words, it acts like ESC 5 ^R.

When you give ^R a negative numeric argument, for example -5, it searches forward for the 5th occurrence of the string. In other words, it acts like ESC 5 ^S.

Try all of these requests with numeric arguments until you feel comfortable with them.

Numeric Arguments with Regular Characters

Whether you realized it or not, regular letters and numbers are actually Emacs requests, too. For instance, you know that ^D means "delete the character at the cursor." What does an ordinary d mean? What happens when you type an ordinary d, or any number, letter, or punctuation mark? It goes into the buffer and appears on the screen. Printing characters (other than #, @, and \), are said to be self-inserting, because if you type one, it inserts itself into the text.

Giving a positive numeric argument to a self-inserting character causes it to insert itself that many times. For instance, if you type:

ESC 24 Q

you see 24 Qs appear on the screen all at once. This is a good way to get lines of dashes, underscores, asterisks, etc.

If ever you type a numeric argument, or are in the middle of typing one, and decide that you did not mean it, type a ^G to get you out of it. The reassuring beep verifies that any possible numeric argument has been discarded.

RE-EXECUTING A REQUEST

^C

After typing a request, you may just want to repeat it once, or to repeat it several times without determining exactly how many times. The ^C request, re-execute-command, lets you re-execute the last keystroke request entered. This request is not especially useful for repeating may of the requests learned so far, since it is just as easy to retype them as it is to type ^C. Some requests, however, require that you type a sequence of keys (e.g., ESC F in the section on word requests); typing ^Cs prevents you from making an error when repeating these. In addition, when you use ^C to re-execute a search request, it does so, reusing the same search string.

MULTIPLE EXECUTIONS OF A REQUEST

^U

A rather special request lets you repeat requests in a couple of different ways. The ^U request (multiplier) multiplies the next request 4 times for each use. For example, ^U^F moves forward 4 characters; ^U^U^F moves forward 16 characters. This request can also be followed by a number, in which case it behaves like ESC followed by a number; ^U13^F moves forward 13 characters and ^U-13^F moves backward 13 characters. A ^U with no following number is considered a positive numeric argument.

Try using ^U with various requests that accept numeric arguments (except ^Y an ^@). If you experiment with it, you will get an idea of the approximate "space" covered by 4, 16, or 64 repetitions of the various requests. Then, when something "looks about that far," you can type an appropriate number of ^Us to get in the right neighborhood, at least.

UNDOING THE ACTION OF A REQUEST

^\

In addition to using negative numeric arguments to reverse requests, you can use the ^\, undo-prefix request as a prefix to a request to reverse the effect of the usual action of that request. For example, ^X_ (described in "Underlining Words") underlines an entire region; ^\^X_ de-underlines an entire region. If a particular request does not accept the undo-prefix, Emacs tells you.

^\ and Self-Inserting Characters

The undo-prefix request can also be used as a prefix to a self-inserting character. In this case, the undo-prefix request searches backwards on the current line for the last occurrence of the self-inserting character, deletes the character, then returns the cursor to its original position. For example, d "inserts" a d at the cursor; ^\d "undoes" the last d you inserted.

GOING TO A SPECIFIC LINE NUMBER

ESC G

A request that lets you move the cursor to a line specified by its line number is ESC G, go-to-line-number. This request is especially useful when editing Multics programs, since many Multics tools give diagnostics in terms of line numbers in their input files. If you happen to be positioned on the first line of the buffer, going to the 241st line is easy; just do 240 ^Ns by typing:

ESC 240 ^N

However, if you are on some other line, it is easier to use ESC G. This request takes a positive numeric argument that is the number of the line to which you want to go. Typing:

ESC 241 ESC G

moves the cursor to the beginning of line 241, no matter what line you are currently on. If that line is not on the screen, Emacs selects the appropriate area of the buffer and displays it. You can also simply type:

ESC G

and you will be prompted in the minibuffer for the line number you wish. End the prompt with CR, as usual. Try it out.

When you are finished, write out your practice file with ^X^S or ^X^W. For the rest of the tutorial, you can log in or out whenever you want a break, and your file will always be there to read in if you want some ready-prepared text to work with. Otherwise, you can create new text, and files, if you prefer.

SECTION 9

WORKING WITH WORDS

WHAT'S IN A WORD

Some of the most useful requests in Emacs are those which relate to words. Even if you are typing computer programs or other non-English text material, the facility to move around word by word, delete words, etc., is very helpful.

The word movement and deletion requests have a deliberate parallelism with the character movement and deletion requests: ^F, ^B are forward character, backward character; ESC F, ESC B are forward word, backward word. Similarly, ^D and # are delete character and rubout character, while ESC D and ESC # are delete word and rubout word.

A "word" in Emacs consists of an unbroken string of upper and lowercase alphabetics (a-z and A-Z), numbers, underscores, and backspaces. Lower and uppercase letters can be mixed in any way. For example, "new_payroll", "zeBra," and "begin" are each one word; "delete-char" and "segname$entry" are each two words. "March, I said," is three words.

It will help in learning the following requests if you picture to yourself the point, rather than the cursor. Think of the point at the left edge of the cursor, between the character at the cursor and the character or space just preceding it.

Also remember that the letters ESC represent the escape key; when followed by a space and a character, you type the escape key and the character, but not the space. Alphabetic characters are given in capitals, but you can type either an upper or lowercase letter. Thus, for ESC F, you should type just the escape key, release it and type an f or F.

MOVING FORWARD AND BACKWARD

ESC F

The forward word request, ESC F (forward-word), moves the cursor forward over one word.

- If the cursor is currently on a character that is part of some word, it moves to the first character after that word.
- If the cursor is currently on a character between two words (even if they are separated by many blank lines, punctuation, breaker bars, etc.), it moves to the first character after the second of those two words.

For example, type in this sentence and position the cursor at the beginning of the line (first case, above):

Yes, it is true.

Type an ESC F, and the cursor, on the "Y," moves to the comma. Now type another ESC F (second case, above). The cursor moves to the space after "it." Reposition the cursor to under the e or s in "Yes," and try again (first case, above). Again, the cursor moves to the comma.

THE ESC F request also accepts numeric arguments. To go six words forward, you type ESC 6 ESC F. Or, you can type ESC F and then ^Cs to move forward a word at a time. This request goes backward with negative numeric arguments.

ESC B

To move backward by words, you use ESC B, backward-word.

- If the cursor is currently on some character of a word other than the first, it moves to the first character of that word.
- If the cursor is on a character between two words, or on the first character of a word (the point would then be between two words), it moves to the first character of the preceding word.

Put your cursor under the t in "it" in the sentence just typed, and do an ESC B (first case, above). Now, with the cursor under the i in "it," try again (second case, above).

This request also accepts a numeric argument, moving backward the specified number of words (or forward, if the argument is negative).

The complementary use of ESC F and ESC B is well illustrated by the problem of adding parentheses to "Tony" in the following:

Anthony Tony Burns.

If the cursor is at the period, the sequence:

ESC B ESC B (ESC F)

does it. Try it out for yourself.

DELETING WORDS

ESC #

Deleting words is perhaps the second most common editing operation (after deleting characters) when entering text. To delete the last word you typed, i.e., the word to the left of the cursor, you use ESC #, rubout-word. It deletes the word to the left of the cursor, or deletes that part of the word to the left of the cursor if the cursor is in the middle of a word. The action of ESC # is best described as though it were doing ^@ (setting the mark), then ESC B (backward word), and finally ^W (wiping the region). That is to say, ESC # removes all text between the cursor's starting point and where it winds up after an ESC B. Note, however, that the mark is not really set.

To summarize:

- If the cursor is immediately after a word, ESC # deletes only the characters of the word it follows.
- If the cursor is in the middle of a word, ESC # deletes that part of the word to the left of the cursor.
- If the cursor is at any other point, ESC # deletes all characters between the cursor and the preceding word, and that preceding word (intervening punctuation and spaces are deleted, too).

In your sentence, "Yes, it is true," put the cursor under various letters and try to predict what will be deleted before doing ESC #s. Be sure to try with the cursor under the i in "it."

Successive ESC #s remove words farther and farther back. This request does kill merging, so if successive words, and the punctuation and white space between them, are deleted by ESC #, one ^Y retrieves the whole deleted area as it initially stood. A numeric argument can also be used with ESC #, deleting backward the specified number of words (or forward, if the argument is negative).

ESC \177

Just as the # and the rubout key, represented as \177, had the same effect on characters, the ESC # and ESC \177 requests have the same effect on words. You can use these two requests interchangeably. The command name for ESC \177 is also rubout-word.

ESC D

Forward word deletion is performed by ESC D, delete-word. It deletes the word, or part of a word, to the right of the point, i.e., to the right of, and including the character at, the cursor. It deletes forward from the point where the cursor is, to the place where ESC F would go.

- If the cursor is on the first character of a word, ESC D removes the entire word.
- If the cursor is in the middle of a word, ESC D removes all the characters from the one at the cursor to the last character of the word.
- If the cursor is between words, ESC D removes all white space and punctuation up to the second word, and the second word.

Consider the sentence:

We have no melons today, Mrs. Johnson.

with the cursor under the r of "Mrs." To replace "melons" with "pears", you type: ESC B ESC B ESC B ESC D and then the word "pears." The ESC D request accepts numeric arguments. Again, try these requests out with your own text by predicting the action before typing the request sequences.

CAPITALIZATION

ESC L, ESC U, ESC C

A unique set of capabilities is provided by three requests that control the "case" of words, i.e, lowercase (jack), uppercase (JACK), or capitalized initial letter (Jack). These three requests are:

- ESC L to lowercase a word (lower-case-word)
- ESC U to uppercase a word (upper-case-word)
- ESC C to capitalize the initial letter of a word (capitalize-initial-word)

Each of these requests can be issued with the cursor either:

- on any character of a word, or
- immediately after a word to alter its case.

For example, type:

thomas

and leave the cursor right after the "s." To capitalize the initial character, merely type an ESC C, and you have:

Thomas

The cursor is always left immediately after the word whose case is transformed. If you wish to capitalize the initial characters of several words, say:

thomas alva edison

you can move the cursor to any letter of the word "thomas," type ESC C, leaving you on the space after "Thomas," ^F to go to "alva," another ESC C, leaving you between "Alva" and "edison," and ^F and a final ESC C, leaving you after "Edison."

All three word-case-altering requests leave the cursor immediately after the word whose case is altered. Since that position is a good place from which to issue such a request, you can position the cursor after "Thomas" and type, ESC U ESC L ESC C ESC U etc., and watch THOMAS, thomas, Thomas, THOMAS, etc., replace each other on the screen while you decide which form you like.

These requests deal with all the characters in a word. Thus, a word like "MaGicAl" can be converted to "Magical," "magical," or "MAGICAL" by use of these requests. Also, if you hold a shift key down too long, ESC C can easily change, for example, "MAgical" to "Magical" when you finish typing the "l."

Each of these requests also accepts a numeric argument. With a positive argument, they affect the next specified number of words. If the cursor is currently within a word, that word counts as one of the specified number. With a negative numeric argument, these requests act on the specified number of previous words.

- If a word-case-altering request is issued between words, but not immediately after the first one, it alters the case of the second word and leaves the cursor after it.

Changing the Case of Regions

^X^L, ^X^U

Two similar requests that also alter case are ^X^L (lower-case-region) and ^X^U (upper-case-region). These two requests operate on regions; all letters within the limits of a region are made lowercase by ^X^L, or uppercase by ^X^U. You must, of course, set the mark first, move the cursor to the other end of the region, and then issue the appropriate request. For example, to add emphasis to this sentence:

The boy survived for sixteen days in the desert.

you could set the mark at the "S" of "sixteen," or the space before it, move the cursor to the space after "days" (or even to the "i" in "in"), and type ^X^U. The sentence would then appear like this:

The boy survived for SIXTEEN DAYS in the desert.

For these two requests, the cursor remains where it is, i.e., the cursor is not left after the region whose case is changed (as it is left after the word with ESC U or ESC L). However, you can type these requests one after the other to see which result you like best, since the region remains the same. You can also make use of ^X^X to verify the limits of your region.

UNDERLINING WORDS

Related to the word-case-altering requests are the underlining and underline-removing requests. They cause a word to be underlined, or remove the underlining from an underlined word. Most current video terminals either do not have the ability to underline text at all, or can only do it in a very limited way. Therefore, underlined text in Emacs may appear as:

```
H\010__\010e_\010l_\010l_\010o
```

where "Hello" is wanted. The \010s are backspaces; they are shown in this way because almost no video terminals can overprint characters, even among those with a limited underlining capability. The text in your buffer that will be written out to your file actually contains the proper number and placement of backspaces, even if the appearance is disconcerting. However, to avoid problems with dprinting text or editing it with other editors, do not use backspaces for underlining; use ESC _ instead.

ESC _

Typing in backspaces in order to underline words is confusing and error-prone, especially on a video terminal. Thus, Emacs provides ESC _, underline-word, for automatically underlining words. To use this request, position the cursor to:

- any place within a word to be underlined, or
- immediately after the word

just as for the word-case-altering requests. The ESC _ request then underlines the word correctly, leaving the cursor immediately after the word. In order to get "begin," for instance, type b,e,g,i,n, and then ESC _.

^Z_

The ^Z_ request, remove-underlining-from-word, removes the underscores and backspaces from an underlined word. Again:

- the cursor can be at any point in the word, or
- the cursor can be immediately after the word.

Since this request also leaves the cursor immediately after the "deunderlined" word, successive ESC _ and ^Z_s add and remove underlining from the same word, in alternation.

Underlining Regions

^X_

You can underline an entire region in one keystroke by issuing the ^X_ request, underline-region. When given any numeric argument, ^X_ removes the underlining from a region. White space within the region is underlined (or "deunderlined") by this request if you set the ESC X opt underline-whitespace option to "on" (see ESC X opt).

LOCATING WORDS

In addition to searching for and locating a given sequence of characters, Emacs can also locate words. In the sentence:

> Yes, I know, Miss Smith's theater
> is the One for me

Assume that the cursor is on a previous line or in the word "Yes," and you want to find the word "is." With ^S, the string search request, prompting with the string "is" gets you to the "is" in "Miss." You could, of course, keep repeating ^Ss until the right occurrence is located (or search for the string "er^QCRis").

^XW

However, the word searching request, ^XW (multi-word-search), locates a word. Type in the above sentence, if you have not already, and position the cursor as suggested. Now type a ^XW. Emacs prompts:

> Word Search:

Then type in the word, is, and a CR. The cursor is left immediately after "is."

This word search request finds words regardless of capitalization or underlining. You can locate "One" or "me" in the above sentence by providing "one" or "me" in answer to the prompt.

This request can also find sequences of words, which is to say, several sequential words, separated by any amount of punctuation or white space. If the cursor were several lines above the sample sentence, you could find it by answering ^XW's prompt:

Word Search: i know miss smith CR

The cursor is left after the "h" in "Smith." Punctuation and capitalization make no difference. If you search for the sequence "theater is the," you will see that white space is also ignored.

Actually, punctuation and white space are not really ignored, but they are treated the same, as separating one word from another. Thus, the sequence:

^XW jack knife CR

locates:

jack knife
Jack, knife
Jack... "KNIFE

but not

jackknife

which is one word, not two.

LOCATING WORDS BY THEIR PREFIX WITH *

The word search request can also locate words by searching for words that start with a given string. This is useful for searching for long words. To indicate that a word-prefix is to be searched for, type the first letters of the word followed by an asterisk (*). For example, type a ^XW, then anted*, and the escape key to search for "antediluvian" or "antedated." You can use word-prefixes in this way as part of a word sequence being searched for, as well. To search for The "Antediluvian" Era, for example, you could answer the prompt with:

the anted* era CR

Like most other Emacs search requests, typing just a CR to its prompt uses the last search string provided. The search proceeds from the current point in the buffer to the end of the buffer.

There is no reverse word search, but if you supply any positive numeric argument to ^XW, e.g., ESC 1 ^XW or ^U^XW, the search begins at the beginning of the buffer. Because this request checks for so many things, it can be slow. Therefore, if you know what exact characters you are looking for, ^S or ^R is faster. If you know the words, but not the case, intervening punctuation, etc., use ^XW.

SECTION 10

MANIPULATING SCREENS AND BUFFERS

MOVING THROUGH A BUFFER SCREEN BY SCREEN

Often you wish to "page" through a document, reading through it, or glancing over it to locate a certain section to read or edit. This is accomplished on Emacs by paging through the text screen by screen. You need to be able to see succeeding windows to do this.

^V

In order to view the next screen, you issue the ^V request, next-screen, read in your longest practice file and "^V through it." Each time you strike a ^V, the cursor is left at the upper left corner of your screen; not only does the window fill with new text, but the cursor also moves to a new place in the buffer. If you type a ^P after a ^V, Emacs chooses a different portion of the buffer to display, centering the line of interest.

With a positive numeric argument, ^V pages forward the specified number of next screensful, and displays it. With a negative numeric argument, it pages backward the specified number of screensful (previous screens).

Note that as you type these ^Vs, the first line on the new screen is always the same as the last line on the old screen. This helps orient you as you go through the text.

ESC V

You can also page backward through a buffer. The two-key sequence, ESC V, lets you view the previous screen. With this request, the first line of the old screen is displayed as the last line of the new screen. The command name is prev-screen.

With a positive numeric argument, ESC V moves backward the specified number of previous screensful and displays it. With a negative numeric argument, it pages forward the specified number of screensful (next screens).

MOVING TO EITHER END OF A BUFFER

Now you can go forward and backward through the buffer character by character (^F, ^B), line by line (^N, ^P), word by word (ESC F, ESC B) and windowful by windowful (^V, ESC V). However, as with lines, you often need to get to the beginning or end of a buffer. What ^A and ^E do for lines, ESC < and ESC > do for buffers.

ESC <

The request for going to the beginning of a buffer is ESC <, go-to-beginning-of-buffer. Think of the less-than sign as an arrow pointing to the buffer's beginning. The ESC < request displays the first windowful of the buffer and puts the cursor at the very first character.

ESC >

The ESC > request, go-to-end-of-buffer, displays the last windowful of the buffer and puts the cursor after the last character or newline character (if the buffer ends with a newline) in the buffer. If, after typing ESC >, you see the cursor on a line by itself, that means that an empty line, one with only a newline in it, is at the end of your buffer. If the cursor does not go onto a line by itself, you should type a carriage return so that the file does end in a newline.

You can, of course, get to any line between the first and last with the other requests learned, especially ^P or ^N with numeric arguments, or ESC G when you know the line number desired.

EDITING MORE THAN ONE BUFFER

In an Emacs session, each use of the find file request, ^X^F, results in a new buffer. The ^X^F request prompts for a filename (pathname of a file), terminated by a carriage return.

If no existing buffer contains that file yet, ^X^F reads the file into a buffer, names the buffer by the first component of the entry portion of the filename (e.g., names the buffer "first" when the filename is >udd>Sales>Smith>first.practice), and sets the default file of this buffer to the file just read.

If one or more buffers containing the named file do exist, they are listed on the screen (see "Listing the Buffers," below). You are then prompted to specify the buffer you wish to use:

Buffer:

Type a name and end the prompt with a carriage return. If you type one of the listed buffer names, ^X^F switches to it and its version of the file. If you decide instead to use a new buffer, give a new buffer name, and ^X^F reads the file into it and gives this buffer the new name. A blank response for the buffer name reuses the buffer named for the first component of the entry portion of the filename.

For example, the first time you read in a file named ibm.data, it goes into the buffer named ibm. If you read that file again for some reason, Emacs lists the buffers containing the file, only one in this case, the buffer named ibm, and prompts you for a buffer name. You decide to use a new buffer for this copy of the file, and type "new."

If you read ibm.data in a third time, Emacs lists both "ibm" and "new" as buffers containing the file (though they may contain different versions of ibm.data if you have edited them). You can use yet another buffer by typing a different name, or reuse one of these two by typing its name. If you simply type a carriage return, Emacs reuses "ibm."

Going from One Buffer to Another

^XB

When you have several buffers containing many different files, or containing versions of the same file, you often need to switch from one to the other. The request that does this is ^XB (when typing this request, you must release the control key before typing b, or you will end up with the ^X^B request described below). The ^XB request, select-buffer, prompts for the name of the buffer to which you want to go. Type the buffer name and a carriage return.

If the buffer exists, Emacs refreshes the screen with the last windowful that you were editing in that buffer, and the cursor is placed at the same point where it last was.

If the name given to the prompt is not that of an existing buffer, Emacs creates such a buffer, and displays it on the screen (you see an empty window, since the buffer is empty).

If you respond to the prompt by typing only a carriage return, you return to the last buffer you were in before entering the current buffer.

When you switch buffers with ^XB, the mode line changes to reflect the name and mode of the buffer switched to. The path line also changes to let you know the pathname of the file that was read into this buffer, or last written out from it.

Listing the Buffers and Local Displays

^X^B

To list the buffers in use in an Emacs session, issue the ^X^B request, list-buffers. This request displays a list of buffers as a local display. A local display consists of information displayed at the top of your screen that temporarily replaces the text being edited. A line of dashes and stars, like this:

```
--* * * * * * * * * * * *--
```

is also displayed so that you can tell that your buffer has not been destroyed, but simply that a local display is being shown "in front of" your text.

The local display for ^X^B contains the name of each buffer and the pathname of the file in it, if any. For some buffers, two symbols, greater-than sign (>) and asterisk (*), appear to the left of the buffer names. The greater-than sign indicates the buffer you are currently editing. An asterisk indicates a modified buffer, i.e., a buffer that contains modifications or additions that have not yet been written out. Only when no buffers have an asterisk beside their names will ^X^C let you exit Emacs without the query, "Modified buffers exist. Quit?"

When you finish viewing the list of buffers in the local display, you want to remove it from the screen. If you type any editor requests, the local display vanishes and is replaced by what was there before it. However you may not want to issue any such requests when you cannot see the cursor's location.

A request is provided for just such circumstances. It is the "no operation" request that does nothing at all! Since typing any editor request removes a local display, this request can be used to do so without doing anything else. You issue it by striking the linefeed key on your terminal (this is the same as ^J, noop, on all terminals).

Sometimes, local displays take more than one screen. In this case, the last line of the screen says:

--More?-- (Space = yes, CR = no)

If you see this, hit the space bar once to see each successive screen of the local display. The last screen of the local display is indicated by the line of dashes and asterisks; you restore the buffer to the screen with linefeeds, when ready. If, during a multiscreen local display, you decide you have seen enough, typing a carriage return (CR) instead of a space terminates the display and restores the buffer to the screen.

A GARBLED SCREEN

Occasionally, you may not believe what you see on the screen. Sometimes bad telephone lines, or unexpected messages from Multics, or something you just do not understand may cause the screen's contents to become invalid. This may be due to hardware problems, bugs in new versions of Emacs, or bugs in your terminal. At any rate, you need to clear the entire screen and put it back the way it ought to be.

^L

This is accomplished with the redisplay request, ^L, redisplay-command. Try typing ^L; the screen clears and is refilled, with the cursor in the middle of the screen (unless you are at the top of the buffer, in which case the cursor is at the top). On fast terminals, ^L can be used to reposition the window so that the line with the cursor on it is at the middle, or to remove a local display (using ^L for either of these purposes is not so useful on slow terminals).

With a numeric argument, ^L redisplays and repositions the window so that the line with the cursor appears at a place of your choice in the window. A positive argument gives the number of lines below the top of the window that you want the cursor's line to be, where the top line is 1 (or 0). For example, ESC 1^L moves the current line to the top, ESC 6^L moves it six lines from the top, etc. With a negative numeric argument, the cursor's line moves to the specified number of lines above the bottom of the screen, where the bottom line is -1. For example, ESC -2 ^L moves to two lines from the bottom, etc.

MARKING AN ENTIRE BUFFER

^XH

The most common reason for marking an entire buffer is that you want to move the text and insert it in another buffer. The ^XH request, mark-whole-buffer, sets the mark at the end of a buffer and the point (cursor) at the beginning. This marks the whole buffer, although the linefeed at its end is not in the marked region. A ^W would delete it. The sequence, ^XH ^W ^XB... go to place in new buffer where you want the marked buffer... ^Y, effectively moves an entire buffer.

KILLING AN ENTIRE BUFFER

^XK

The ^XK request, kill-buffer, kills an entire buffer. You are prompted for the name of the buffer to be killed, and end the prompt with a carriage return. Buffers are usually killed to conserve storage or to remove them from buffer listings given by requests like ^X^C or ^X^B. When you try to kill the current buffer, you are asked for a new buffer to go to.

SUMMARY OF TERMS

Two new terms for you to learn are:

- local display
- redisplay

SECTION 11

HELP

By this point you may be having trouble keeping track of all the requests learned so far. However, whenever your memory of a particular request lapses, help is at hand!

WHAT DOES THIS KEY DO?

ESC ?

When your are not sure what a given key does, you can type the ESC ? request, describe-key. This request displays the documentation for the request you are unsure of. For example, to find out what ^V is and does, type ESC ? Emacs prompts in the minibuffer:

Explain Key:

Now actually type a ^V, in the usual manner. A description of what ^V does appears as a local display at the top of your screen:

```
^V      next-screen

Display next screenful of this buffer.  Leave cursor
at upper left hand corner of screen.

 --* * * * * * * * * * * *--
```

The command name associated with the ^V request is next-screen (Emacs refers, in prompts and Emacs-produced documentation, to requests as "commands"). The documentation describing next-screen follows the line giving the key name and command name.

To clear away the local display describing ^V, hit linefeed a couple of times. Try ESC ? with a few more requests.

When you forget what request a given key invokes, or need to find out what you just did by accidentally typing the wrong request. You can ask for the command name invoked by a given key without the documentation to save time. By giving ESC ? a numeric argument, e.g., ESC 1 ESC ?, you get this prompt:

Show Key Function:

Then type the key in question, say, ^W. Emacs responds in the minibuffer:

^W = wipe-region

EXTENDED REQUESTS

Some requests are issued by a single keystroke, such as ^D, which invokes the delete character request, delete-char. Other, less common ones, are invoked by two-key sequences beginning with ESC, such as ESC ? for describe-key. Still less common requests are invoked by two-character sequences beginning with ^X or ^Z (e.g., ^X^W). Requests that are the least common have to be invoked by actually typing in their command names.

Though some requests are "less common", they are no less important. They require more keystrokes simply because you have less occasion to use them. Single keystroke requests are reserved for those tasks that you perform often while editing.

The requests known as "extended requests" are those invoked by typing their command names to Emacs. An example of an extended request is "fillon," which enters "fill mode" in a buffer. Fill mode sets up a buffer so that you do not need to worry about the ends of lines when typing text, and never need to type carriage returns (except, of course, when ending prompts, or when you want explicit control over the format and line-breaks of your document). Fill mode is ideal for typing in text from a written page, or composing a document spontaneously. You just keep typing; the lines get broken automatically.

To invoke an extended request such as fillon, you clearly cannot just type "fillon." If you did, it would simply go into the buffer like any other characters. You have to let Emacs know that the next characters are the name of an extended request.

ESC X

The request that notifies Emacs to expect an extended request's command name is ESC X, extended-command. Type this two-key sequence. Emacs prompts in the minibuffer:

Command:

Now type the word "fillon" (no quotes, just the six letters), and a carriage return. The name of the "fill" minor mode appears in the mode line after the name of the current major mode, Fundamental.

In general, you invoke an extended request in this way:

- type ESC X
- type the name of the extended request
- if the request takes any arguments, type a space and then the argument(s)
- type a carriage return

WHAT KEYS DO THIS JOB?

apropos

A very important extended request provides you with all the command names, and their associated keys, that relate to a given topic. It is used if you remember something about a particular request, but you cannot remember the key that invokes it, or what its command name is. The apropos extended request finds all requests that have a given character string in their command name, and provides a local display telling you what keys invoke them. The topic that you are interested in is typed, after a space, as an argument to apropos.

For instance, suppose you forget which request goes to the end of a line. Choose a topic you think is appropriate, since apropos must have an argument. If it does not, you get this error message:

Wrong number of arguments to extended command apropos.

In this case, "end" seems a reasonable choice. So, type ESC X apropos (no spaces yet), a space, the argument (end) and a carriage return. The apropos request displays the command names of all requests available in this buffer whose names contain the character string "end." You see ^E (go-to-end-of-line), ESC > (go-to-end-of-buffer), a few surprises, such as ^XM (send-mail), and others. Once you learn from apropos what requests are available, you may be jolted into recognition ("Right, ^E is go-to-end-of-line!"), or you may need more information ("Hmm, ^E looks right, but I'd like to know exactly what it does.") You can, in the latter case, type ESC ? ^E to get the full documentation.

The apropos request also lists all relevant extended requests if their names contain the specified character string. However, you cannot use ESC ? (describe-key) to find out about extended requests, since ESC ? prompts for a single key, and then describes it. If you try to type "fillon" to ESC ?, for example, it reads the "f" and tells you that "f" puts an f into the buffer. The remaining letters, illon, would go into your buffer.

WHAT DOES THIS EXTENDED REQUEST DO?

describe

To find out, then, what an extended request does, you need another help request. The describe extended request retrieves the documentation for extended requests. Like apropos, it also requires an argument; in describe's case, the argument (separated by a space from the last character of "describe") is the command name of the extended request you are interested in.

For example, to find out about fillon, you type ESC X, describe, space, fillon, carriage return. The documentation for fillon is then shown on your screen as a local display. Try describe out with the other two extended requests you know, describe itself and apropos.

TANGIBLE HELP

make-wall-chart

If you want help in a more permanent and tangible form than that provided by ESC ?, apropos, and describe, you can try the make-wall-chart extended request. This puts a list of all the currently defined command names and their associated keys into a buffer.

You should then write the contents out to a file and dprint a copy. The list produced in this way, "suitable for framing", is a handy reference that you might want to keep near your terminal. The wall chart produced is 132 columns wide and the requests are divided into three columns. A sample (of one column only) appears below.

```
esc-F       forward-word
esc-G       go-to-line-number
esc-H       mark-paragraph
esc-I       indent-relative
esc-K       kill-to-end-of-sentence
esc-L       lower-case-word
```

MORE HELP AND WHAT DID I JUST DO?

^_

The ^_ request, help-on-tap, provides all of the above forms of help and some additional forms, depending on the character that follows it. (On some terminals, you may have to type ^? to send the ^_ character.) A ^_? displays the current repertoire of ^_. A ^_H shows where to get more help. The ^_A and ^_D requests are shortcuts for the ESC X apropos and ESC X describe extended requests; they work in the same way as those two extended requests, respectively. A ^_C works just like ESC ?, describe-key. A ^_^G does a ^G, as usual, in case you want to get out of help-on-tap.

Sometimes when you are happily editing away, things happen that you do not expect. If you want to track down your error (reconstruct the scene of the crime, as it were), you can get a local display of the last 50 characters typed in. The ^_L request provides this, and is very useful as a learning tool. It lets you examine what you did so that you can identify and correct any problems.

SECTION 12

SENTENCES AND PARAGRAPHS

Besides being able to recognize and manipulate words, Emacs also performs useful manipulations on sentences and paragraphs.

SENTENCES

Emacs can go to the beginning or end of a "sentence" (ESC A, ESC E), and can kill sentences either backward (^X# or ^X DELETE key) or forward (ESC K). In order to do these editing tasks, it must be able to recognize what a sentence is.

Basically, to find a sentence, Emacs looks for period (.), question mark (?), or exclamation point (!) followed by at least one space or tab. Capital letters have no meaning; Emacs sentences need not start with one. If you have numbered items, e.g.:

1. Measure one cup of flour
2. Add a teaspoon of baking soda

Emacs considers "1." to be the end of a "sentence" (or possibly a complete sentence depending on preceding text). The period is the last character of that sentence. The second sentence starts with the first printed character following the end of the first sentence, and contains "Measure one cup of flour," a newline, the number "2," and the period. The third sentence starts with the "A" of "Add." The end of this third sentence does not appear above. However, if the line following this is blank, then "soda" ends the sentence; "a" is the last character in it.

Emacs also considers itself to be at the beginning of a sentence if the cursor is at the beginning of a buffer, and at the end of a sentence if the cursor is at the end of the buffer.

So, to summarize: Emacs actually defines a sentence by its ending. Sentences start with the first printed character after a previous sentence's end. Sentences end with:

- a period, question mark, or exclamation point that is followed by white space
- the last character preceding an empty line
- the buffer's end

Conversely, sentences begin with:

- the first printed character following a period, question mark, or exclamation point that is followed by a space
- the first letter following an empty line
- the first letter at the buffer's beginning

Moving Forward or Backward by Sentences

ESC A

The ESC A request, backward-sentence, moves the cursor to the first character of the current sentence, i.e., if the cursor is on a character in a sentence, it goes to the first character of the sentence. If the cursor is already at the beginning of a sentence, it moves to the first character of the preceding sentence. If you supply a numeric argument to ESC A, the cursor moves backward the given number of sentences.

ESC E

The forward-sentence request, ESC E, moves the cursor to the end of the current sentence, leaving it right after the sentence. If the cursor is already at the end of a sentence, it goes to the end of the following sentence. Anything defined to be the end of a paragraph (see "Paragraphs" below) is automatically the end of a sentence as well. This request also accepts numeric arguments.

Experiment with these two requests until you feel comfortable with the concept of Emacs sentences.

Killing Sentences

^X#

Remember what ESC # did for words? Well, ^X#, kill-backward-sentence, does it for sentences. This request kills backward from the point to the beginning of the sentence, i.e., the first letter following the end of the previous sentence. Thus, in the following text, start with the cursor under the question mark:

```
1. Measure one cup of flour. 2. Add
an egg. 3. Mix well

Voila  You have noodle dough. What
kind of noodles, you ask?
```

After the first ^X#, the question mark, with the cursor still under it, is located where the "W" of "What" was, and that sentence is gone. The ? remains because it was not included in the text between the point and the beginning of the sentence. Subsequent ^X#s leave the ? and cursor where the "Y" of "You" was, the "V" of "Voila," the "M" of "Mix." Note that the blank line is deleted as simply being so much white space between sentences. It is this blank line, however, that makes "Voila " be a sentence on its own, rather than a continuation of the "Mix well" sentence, even though no end-of-sentence punctuation follows "well." The next ^X#s puts you at the former position of "3," the "A" of Add," and so on.

Sentences killed successively with ^X#s merge on the kill ring. One ^Y retrieves them, as well as the intervening white space, so your text looks the same as it did. This request also accepts a numeric argument, killing the specified number of sentences backward and entering them on the kill ring.

^X\177

The ^X\177 request (type a ^X and then the delete key) is also kill-backward-sentence, and behaves the same way as ^X#.

ESC K

As promised at the beginning of this section, Emacs kills sentences forward via the ESC K request, kill-to-end-of-sentence. This deletes text going forward from the cursor to the end of the current sentence. If you are at the end of a sentence, e.g., the cursor is on the space immediately after a period, this request deletes forward to the end of the next sentence. Sentences killed successively are merged, and can be retrieved with a single ^Y. Also, ESC K accepts a numeric argument, killing forward the specified number of sentences; these sentences enter the same slot on the kill ring.

Try ESC K out with the sample sentences above. If you typed them in before, and killed them with ^X#s, you can, of course, retrieve them with ^Y to avoid retyping.

PARAGRAPHS

Emacs defines a paragraph in one of two ways, both controlled by the "paragraph-definition-type" option of the ESC X opt extended request (see Section 17). When this option is set
* to 1, a paragraph is defined as being all text between two blank lines. In other words, a paragraph begins immediately after a blank line, and ends with the last character preceding another blank line. If, after a blank line, you type spaces or a tab before entering text on the line, that white space is still considered part of the paragraph (unlike white space preceding a sentence).

When the option is set to 2, as it is by default, an indented line starts a paragraph, i.e., any spaces or tabs at the beginning of a line begin a paragraph, and the paragraph ends with the last character on the line preceding the next indented line.

With either option setting, a control line for the runoff or compose Multics commands (e.g., a line containing only .sp or .spb) is a paragraph all by itself. Paragraphs also begin with the line following a "control" paragraph. Compose control .unh (hanging undent) is special-cased so that paragraph-recognizing functions consider the second line after .unh to be the first line of the paragraph acted upon (e.g., ESC Q, runoff-fill-paragraph, does not merge the line following .unh with subsequent lines forming the paragraph to be filled).

By the above definitions, then, a paragraph always begins at the beginning of a line, and the lines must be:

Type 1

- preceded by an empty line
- a runoff or compose control line
- the beginning of the buffer
- preceded by a runoff/compose control line

Type 2

- indented
- a runoff or compose control line
- the beginning of the buffer
- preceded by a runoff/compose control line

Moving Forward or Backward by Paragraphs

ESC [

The ESC [request, beginning-of-paragraph, moves the cursor to the beginning of the current paragraph. If you are already at the beginning of a paragraph, you move to the beginning of the previous paragraph. This request accepts a numeric argument and moves back the specified number of paragraphs.

ESC]

The request for end-of-paragraph is, as you might expect, ESC]. This moves you to the end of the current paragraph, or the end of the next paragraph if the cursor is already after the last character of paragraph. You can use a numeric argument to move forward many paragraphs. Both ESC [and ESC] can be tried out with the sample recipe, since you have two paragraphs (the blank line separates them because they are type 1 paragraphs by default). With this type of paragraph, you can type in text, spacing your paragraphs one or two lines apart for an appearance both practical and pleasing. Each paragraph can be indented as well, but the empty lines determine the paragraph breaks, unless you deliberately redefine paragraphs with ESC X opt.

Marking a Paragraph

ESC H

You may decide to delete or move a paragraph, or rearrange your text by deleting or moving an entire paragraph. To do this you could mark the region with ^@, after getting to the beginning of the paragraph, then ESC] to the end of it and issue the ^W request. However, ESC H, mark-paragraph, makes this easier. It puts the mark at the beginning of the current paragraph, and the cursor at the end. Your region, in this case a paragraph, is marked in one step, and a ^W wipes it out. The paragraph is saved on the kill ring, and can be reinserted where you please with a ^Y.

Formatting a Paragraph

When you just type in text without regard for line breaks, your screen's left edge begins to look pretty messy with all the continuation lines (\c preceding the text). Since this is such a convenient way to enter text, however, Emacs provides a request that "tidies up" your text, paragraph by paragraph.

ESC Q

The ESC Q request, runoff-fill-paragraph, formats paragraphs for you. It rearranges your text so that words are not broken in the middle at the ends of lines. The continuation lines are broken, instead, between words, giving you a neat "ragged right" margin, and no "\c" lines on the left margin. (You can also set your margins; see Section 13.)

Type in about three lines of text without any carriage returns. Type an ESC Q and watch the screen. The text is rearranged automatically. Your new paragraph takes up a few more spaces since the sentences have to be rewritten to accommodate the new line breaks. You can type in text forever, issuing ESC Qs at the end of each paragraph to format it.

If you want an adjusted right margin, issue ESC Q with a positive numeric argument, e.g., ESC 1 ESC Q. To try this with the test paragraph just typed, issue an ESC H, ^W, and ESC 2 ^Y. You have marked the formatted paragraph and deleted it, and yanked the contents of the kill ring's second slot. These contents are the original paragraph, continuation lines and all! That is because ESC Q put the unformatted paragraph into the kill ring when it formatted it, and it was pushed into the second slot by the ^W you just did.

So, ESC Q, without an argument:

- formats the current paragraph with a ragged right margin
- puts the original paragraph into the kill ring

With a positive numeric argument, ESC Q:

- formats the current paragraph with both left and right-justified margins (padding if necessary)
- puts the original paragraph into the kill ring

To set your margins, see ^XF and ^X. in Section 13.

SECTION 13

INDENTATION AND SPACING

Emacs provides numerous requests that deal with indentation and the management of white space. White space is any combination of tabs, spaces, formfeeds, or vertical tabs. Among other things, the Emacs requests discussed here allow you to add, delete, and skip over white space characters, as well as add varying levels of indentation.

BLANK LINES

Adding Them

^O

After you have typed in some text, and perhaps formatted it with ESC Q, you may decide to add another paragraph or two, or an example, or simply a few more lines. The ^O request, open-space, allows you to insert as many empty lines as you wish so that you have a fresh space on the screen in which to work. This request also saves the time it might take on some terminals for Emacs to continually rewrite lines if you were just typing additional text in without first opening a space for it.

Typing a ^O puts a newline into your buffer ahead of the current point. Text after the current point moves down a line and over to the left margin and pushes succeeding lines down one. The cursor remains immediately before the inserted newline, and successive ^Os keep opening up new empty lines. Thus, to insert empty lines above a line of text, issue ^O while the cursor is at the line's beginning. Your cursor is then positioned so that you can issue as many ^Os as desired, and then immediately start typing the new text, since the cursor is right before the new stack of blank lines. With a positive numeric argument, ^O inserts the specified number of newlines. Thus, ^U^U^O, or ESC 16 ^O, opens up 16 lines.

Removing Them

^X^O

If you open up more space than you need, or simply have blank lines that you do not want, ^X^O, delete-blank-lines, removes them.

If the cursor is anywhere on a non-blank line, issuing ^X^O deletes all blank lines after the end of the current line. The cursor is left at the end of the deleted blank lines, i.e., at the beginning of the next non-blank line which has moved up (with all succeeding lines as well) under the line in which ^X^O was issued.

If the cursor is on a blank line when you issue ^X^O, again, all blank lines after the end of the current line are deleted. Thus, you still have one blank line remaining (blank lines above also remain, of course). The cursor is left at the beginning of the next non-blank line.

Try these two requests out for yourself to see how easy it is to open up space, type, and delete the extra.

DEALING WITH WHITE SPACE ON A LINE

Spacing Over Indentation

ESC M

A simple request, ESC M (skip-over-indentation) lets you move the cursor over the white space beginning the current line. The cursor moves to the first non-blank position on the line. This helps get you right to the text after you have issued a ^A, ESC [, or similar request moving you to the beginning of an indented line. In fact, if you are anywhere in an indented line, ESC M moves the cursor to the non-blank at the beginning of the line, so it is frequently more useful than a ^A when working on indented text.

Deleting White Space

ESC \

The ESC \ request, delete-white-sides, deletes all white space surrounding the point on the current line, and closes up the line accordingly. Thus, issuing an ESC \ with the cursor at any of the spaces between "have" and "too" or on the "t" of "too" in the following:

```
I have too much space here.
```

gives you

```
I havetoo much space here.
```

with the cursor left under the "t." Putting the cursor under "I" and typing ESC \ removes the indentation. The point is always left before the first nonwhite space character that followed its prior position.

ESC ^

The ESC ^ request is not issued by typing the escape key and the control key. The "^" here represents the caret character on your keyboard, so you type the escape key and then the caret. This character is commonly referred to in Multics as the "not symbol." ESC ^, delete-line-indentation, deletes all the white space at the beginning of the current line and merges it with the previous line. The cursor can be anywhere in the line when you type the escape and caret key sequence, and it moves to the first nonblank character that began that line. Thus, it usually ends up somewhere in the middle or toward the end of the previous line. Following text moves up to the vacated line.

With a positive numeric argument, ESC ^ does a ^N first, deleting indentation on the next line and adding it onto the current one. In this manner you can connect lines to each other and remove indentation if you decide to change the appearance of your text. For example, to change the following:

```
Mail copies to:
     Bob Burns
     Cindy Hatter
     Jake Voit
```

to:

```
Mail copies to:  Bob Burns, Cindy Hatter, Jake Voit.
```

simply put the cursor on the top line and type (type the characters indicated within the brackets, not the brackets and letters enclosed):

^U ESC^ <2 spaces> ^U ESC^ <,space> ^U ESC^ <,space> ^E <.>

Alternatively, you can start with the cursor on the last line, and type:

ESC^ <,space> ESC^ <,space> ESC^ <2 spaces> ^E <.>

The first way, using a numeric argument, is a little more intuitive.

FILL MODE

Fill mode is a minor mode that can be turned on or off in each buffer. If it is on, i.e., you are "in fill mode," text is broken automatically at the end of each line so that it does not extend past the fill column. The fill column determines the right margin, and is the first column in which text is not to be placed by ESC Q or fill mode formatting. Typing a space, tab, or punctuation mark following a word that passes the fill column signals Emacs to "back up" to the white space preceding that word and break the line there. In addition, the fill prefix, if set, is inserted at the beginning of each new line typed in while in fill mode. The fill prefix determines the left margin, and is empty unless set to contain some combination of spaces and characters (see "Margins," below for changing the fill prefix and fill column). If the fill prefix is not set, i.e., it is empty, the left margin is the left edge of your screen.

While in fill mode, if you do want characters in or beyond the fill column, type a carriage return immediately after the word instead of a space, tab, or punctuation. Or, if you want more than a word, precede each character after the word with ^Q. Alternatively, you could reset the fill column for that line.

Fill mode is an excellent means of entering text without ever having to worry about the ends of lines. Your text is formatted automatically as you type, so you can enter your document rapidly and see how it looks immediately. Use this mode whenever entering English text for maximum convenience.

ESC X fillon AND ESC X filloff

These two extended requests, as their names imply, turn fill mode on or off in a buffer. When entering text, you generally want to be in fill mode, since you can then simply type without ever worrying about carriage returns or line breaks. Your text is formatted as you type.

MARGINS

Your right and left margins are generally set automatically to accommodate your terminal's screen size, thus maximizing your working space. You can, however, change these margins. The left margin can be reset by setting the fill prefix, which is inserted automatically by carriage return, fill mode, and ESC Q. The fill prefix can contain characters and spaces and it "prefixes" all text following a carriage return or formatted by fill mode or ESC Q. Generally, you want only spaces in the fill prefix. The right margin is determined by the fill column, which is the first column in which text is not to be placed by ESC Q or fill mode formatting.

Setting the Margins

^X.

The ^X. request, set-fill-prefix, sets the fill prefix in the current buffer. You position the cursor on a line and type ^X.. Whatever is between the cursor and the beginning of the line becomes the fill prefix. So, if you want a fill prefix of five spaces, put the cursor on the sixth character of a line beginning with at least five spaces and issue this request. A good way to do this is to choose a line indented the way you want it and type an ESC M and then ^X.. Existing text does not change, but subsequent ESC Qs and carriage returns indent text. If fill mode is on, future lines are also indented.

By positioning on a line whose beginning contains non-blank characters, you can include them in your fill prefix. You might want a fill prefix containing asterisks or dashes, for example, to set off a section of your text. To return things to normal, i.e., have no fill prefix, issue ^X. at the beginning of a line (^A gets you out to the left edge before the fill prefix).

^XF

To set the fill column for the current buffer, use ^XF, set-fill-column. The column in which the cursor is located when you type ^XF is set as the fill column. Fill mode and ESC Q use this column as the right margin. When you set it, the value is displayed in the minibuffer like this:

```
Fill column = 65
```

This request accepts an optional positive numeric argument, which is the value to be assigned to the fill column, e.g., ^U 65 ^XF sets column 65 as the fill column and makes column 64 be the last column in which text can be placed when you are in fill mode, or formatting with ESC Q.

If you are dprinting a file on a device that accommodates more characters/line than your terminal, you may want to set the column to an appropriately higher value to save time and paper. Remember, if you do this, that your text on the screen will necessarily include many continuation lines (\c inserted at the beginning of the lines, with words broken randomly).

Centering a Line

ESC S

The ESC S request, center-line, centers the current line between the left edge and the fill column. This is useful for headings and titles, but only if the fill prefix is not set. If it is, you end up with everything "centered" too far to the left. Experiment with ESC S with different settings of the fill prefix and fill column.

MORE ABOUT LINES AND WHITE SPACE

Shearing a Line

ESC ^O

This request, ESC ^O, named split-line, breaks the line at the point. The text beginning at the cursor is moved down a line and indented so that it stays in the same column vertically. The cursor maintains its original position. Thus, you can change:

```
Kit contains:  embroidery thread, stamped cloth
Items needed:  needle, hoop
```

to:

```
Kit contains:
                embroidery thread, stamped cloth
Items needed:
                needle, hoop
```

by putting the cursor at the "e" of "embroidery" and typing:

ESC ^O ^N ^N ESC ^O

Since ^N tries to stay in the same column, it is simple to move the cursor down to the "e" again. The second ^N is all that is necessary in this case since "needle" was already directly below "embroidery." In your editing, however, you will have to make sure the cursor is positioned where you want the line to be sheared.

Now recall what ESC ^ (caret character) does with a numeric argument. It removes the indentation from the next line and tacks the remaining characters onto the current line. So, after you type the sequence above, leaving the cursor two spaces after "needed:", you can simply type ^U ESC ^ to rejoin the bottom lines. A ^P followed simply by ESC ^ rejoins the top two (since ESC ^ by itself removes indentation on the current line and tacks the remaining characters onto the previous line.

Undenting to the Fill Prefix

ESC ^I

If you have an indented line that you no longer want indented, you "undent" it with ESC ^I, indent-to-fill-prefix. This only works if the indentation is caused by something other than the fill prefix, since the request removes leading white space and replaces it with the fill prefix. When the fill prefix is not set, ESC ^I effectively moves the first non-blank character of the line over to the screen's left edge (moving the rest of the line with it, of course). When it is set, ESC ^I simply removes the white space, leaving the first non-blank character after the fill prefix. In either case, the cursor, which can be anywhere on the line when ESC ^I is issued, moves to the first non-blank character.

INDENTATION

Emacs provides a sophisticated method for indenting when you are preparing outlines, programs, tables, and other indented text.

ESC I

ESC I, tab-to-previous-columns, is a very powerful Emacs request. It indents the current point and any part of the line to the right of the current point. The amount of indentation is determined by looking at the previous nonblank line. The printed character following the first whitespace character to the right of the current column determines the new current column. Successive ESC Is line the point up in that manner across the line. For example, starting with the cursor at the beginning of the second line below, the numbers represent the cursor's position after each successive ESC I:

```
Name      Phone      Mint Choc Gran Pnut
1         2          3    4    5    6
```

This request is ideal for building columnated tables. You can type in the first line, spacing your fields by hand, then use ESC I for the remaining lines. Type in an item, ESC I to move to the next field, type that item, ESC I, type, etc. Try creating a table something like the one below:

Name	Phone	Mint	Choc	Gran	Pnut
Harold	867-1066	3	1	0	5
Clumbs	258-1492	5	3	3	2
French	714-1789	0	2	0	0
Revere	417-1775	1	1	1	12

With a numeric argument, ESC I "unindents" one level each time it is invoked. This is especially helpful when creating outlines or similarly structured material.

ESC CARRIAGE RETURN

If you are indenting several lines, you do not have to type a carriage return (CR) after each line and then an ESC I to indent the next. The ESC CR request, cret-and-indent-relative, combines these two functions for you (very useful unless you are in fill mode so you are not typing carriage returns anyway). This request does a carriage return and an ESC I in one step. So, you can end the current line and start a new one indented the same just by typing ESC CR. If the current line is not indented, the new line starts under the first letter of the second word on the original line. Try this out by entering the following:

```
Document preparation should be made as easy as possible.
     (Documents is used loosely here to include memos,
     business letters, theses, professional papers, user
     manuals, instructional booklets, advertising copy,
     and any other informational type of text that we
     need for doing our jobs.)
Emacs helps by simplifying text entry and editing.
```

^X^I

To indent or reindent an entire region by a specified amount, use the ^X^I request, indent-rigidly. It takes a numeric argument that specifies the number of spaces to indent (positive argument), or unindent (negative argument) the region. All lines having any characters within the region defined by the mark and point are indented.

SUMMARY OF TERMS

Some new formatting terms introduced in this section include:

- fill mode
- fill prefix
- fill column

SECTION 14

MOVING BLOCKS OF TEXT

Regions, paragraphs, and buffers can be marked, deleted, and yanked back into new positions whenever you want to move them. The ^@, ^XH, and ESC H requests are the principal means of marking large sections of text that you have learned so far, and ^W deletes any section so marked. The ^Y and ESC Y requests are available for retrieving killed text blocks and inserting them at the cursor. In addition to these requests, Emacs provides some more sophisticated methods of marking and moving blocks of text.

INSERTING AN ENTIRE FILE

^XI

The ^XI request, insert-file, allows you to read additional files into your text. You position the cursor to where you want the file inserted, and issue this request. You are prompted for the pathname of the file you want:

Insert File:

Type the pathname and end it with a carriage return. The word "Reading" flashes on in the minibuffer while the file is read in before the cursor, i.e., to the left of the cursor. The mark is left at the beginning of the inserted text. The cursor, and the text of the original buffer at and following the cursor, are then left after the contents of the inserted file - after the newline at its end (if it ends with one). Your buffer's previous contents are preserved, and the new file becomes part of them. The default file for the buffer remains the same. So, if you originally read in "first.practice" with ^X^F, and insert "second.practice" with ^XI, a ^X^S or ^X^W with an empty prompt response still writes the file out to "first.practice."

With ^XI you can insert as many additional files into the current buffer as you wish, or insert the same file at many places. You can, of course, insert other people's files by giving the appropriate pathname (you must have the proper access). This can be very helpful if several people are working on different sections of a document; you can "assemble" the various pieces by inserting the files in any desired sequence. (Alternatively, you can assemble a document with the runoff or compose command's insert file controls.)

The ^XI request also accepts archive component pathnames. An archive component pathname is a (relative) pathname of an archive segment followed by two colons (::) and a component name. Starnames are accepted, too. For example, to insert Additions.list at the current point, typing ^XI Add*.* does it as long as Add*.* is unambiguous. Stars can also be used in archive component pathnames.

COPYING A REGION

ESC W

Sometimes rather than moving a block of text from one place to another, you simply want a copy of it in one or more additional places in your text. Rather than wipe it with ^W and then yank it back into both the original and secondary position(s) with ^Ys, you can copy it with ESC W and ^Y it into the new position(s) only. ESC W (copy-region), then, copies the region between the cursor and the mark, placing a copy in the kill ring without affecting the original.

SELECTING AND JOINING TEXT ON THE KILL RING

ESC ^W

To join two disparate pieces of text, you can issue the ESC ^W request, merge-last-kills-with-next. This causes the next kill requests, which must follow immediately, to merge what they kill with the last saved kill on the kill ring. Then one ^Y retrieves them as a single entity. For example, typing:

^A ^K ^K ^N ^N ESC ^W ^K ^K

catenates two disjoint lines on the kill ring.

NAMED REGIONS

Emacs provides a way to assign a region to a variable, whose name you choose, so that you can manipulate it at any time during an editing session, i.e., during the same invocation of emacs. A variable is simply a stored region. Variables are maintained for the entire session, so you can work with several named regions at once, instead of the single one defined by the mark and point.

Storing the Region to a Variable

^XX

The ^XX request, put-variable, stores a region away by name. To use this request, you set the mark and point around your chosen region, and type ^XX. Emacs prompts you for a name to be associated with the stored region:

Variable:

Type whatever name you choose, ending with a carriage return. The region then disappears from the screen, and the cursor returns to the character which preceded the region. You have not lost this text; it is merely stored away, and is retrievable by name. You can store as many regions in this manner as you wish. However, be careful not to duplicate names, because Emacs overwrites a previously stored region if you try to store another one in the same variable.

Inserting a Variable

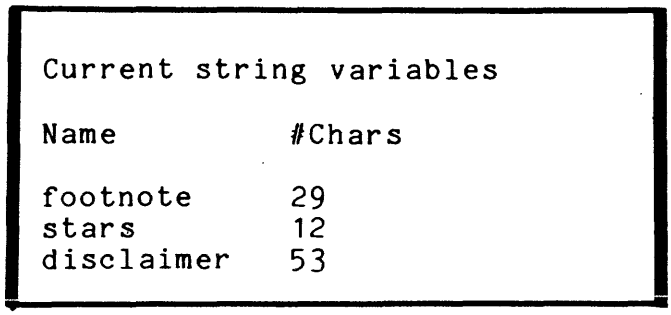
^XG

You use ^XG, get-variable, to retrieve or insert a stored region at the cursor. You are again prompted for the variable name, and the named region reappears in your text at the cursor after you type the carriage return. The cursor is put after the reinserted region, and the mark is set before it. You still have this same region stored as a variable, however, so you can move the cursor and issue additional ^XGs to get more copies into your text.

Listing Your Variables

ESC X lvars

If you forget what your variable names are, or you want to check them to avoid duplication, the ESC X lvars extended request lists them for you in a local display. It also provides the length of each variable, which may give you a clue to what is in them if you cannot remember. The display looks like this:



You can type a linefeed (^J) or resume editing to get rid of the local display.

NAMED MARKS

You can also assign names to marks in much the same manner as you assign them to regions as variables. This can be extremely helpful for setting up a series of places in your text to which you want to return for some reason. Perhaps you want to doublecheck several items, but do not wish to keep interrupting your text entry or editing. Or you may be considering a format change that will affect several parallel points. Even if you simply have several "rough spots" that you know will require further work, you can return quickly and easily to any spot where you have set a named mark. However, named marks, unlike variables, are valid only in the buffer in which they are set.

Setting a Named Mark

^Z^@

The ^Z^@ request, set-named-mark, sets a named mark at the current point. You position the cursor, type ^Z^@, and respond to the prompt:

Mark Name:

with whatever name you choose, terminated by a carriage return. The minibuffer prints a message telling you that the mark is set and giving its name. This named mark is not the effective mark, but is remembered by its name while you set, or set and name, other marks. To go back to this mark, use ^ZG, the go-to-named-mark request. To delete a region that begins at this named mark, you must reset the effective mark at this point, since ^W always wipes out the region between the effective mark and the cursor. Remember, too, that if you change buffers, you can reuse the names assigned to marks set in different buffers, since ^Z^@-created named marks are valid only in the buffer in which they were set. If you do not change buffers, reusing a name reassigns the name to the current mark.

Going to a Named Mark

^ZG

The ^ZG request, go-to-named-mark, moves the cursor to the point where the named mark was set by ^Z^@. You are prompted for the name. Use this request to return to any point where you have previously set a named mark.

Listing Your Named Marks

ESC X list-named-marks

The ESC X list-named-marks extended request is provided for the same reasons as ESC X lvars. It lists the line number and the name for each named mark in a local display that looks like this:

```
Line #     Mark name

5          snow
12         white
53         disclaimer
```

SUMMARY OF TERMS

You can move text by several means now, including inserting entire files and copying regions. You can also move text, or simply move around, by assigning names to regions and marks. The following terms are important for you to remember:

- variable
- named region
- named mark

SECTION 15

KEYBOARD MACROS

A keyboard macro is a sequence of requests that are performed, in order. when you issue a macro-executing request. It is a "mini-program" that you devise to perform some editing task that you must do several times. You can create and use a macro, and then create a new one, or you can save your macros for later use by assigning names and keys to them. The most important thing to know about a macro, though, is that it is easy to create one. All you have to do is tell Emacs that you're about to issue some requests which it should remember as a macro, then issue the requests, and finally, tell Emacs that the macro is finished.

CREATING A MACRO

When would you create a macro? Well, suppose you are preparing a document and decide that you want a section of text to be highlighted by a column of leading asterisks and doublespaced. You can write a macro that inserts the asterisks and blank lines for you.

^X(AND ^X)

The ^X(request, begin-macro-collection, is the first thing you type to start writing a macro. When you type it, Emacs begins "remembering" your next keystrokes as a macro. So, before typing it, you should always define your problem and decide what Emacs requests you would issue to correct it once (you could actually make the correction one time to verify this). Then type ^X(. This appears in your mode line, after the name of the major mode:

<Macro Learn>

Macro Learn is a minor mode, entered via ^X(. Now everything you type is remembered as part of your macro as well as having its normal editing effect on the buffer. In addition, macro definition ends if any request encounters an error during the macro definition. When you type the ^X) request, end-macro-collection, definition of the macro ends and the "Macro Learn" message disappears.

Thus, to insert the asterisks and blank lines you would define this macro:

^X(^A * <2 spaces> ^E ^O ESC 2 ^N ^X)

- ^X(begins the macro definition.
- ^A puts you at the beginning of the line.
- * and two spaces following inserts an asterisk and two spaces before the rest of the line.
- ^E puts you at the end of the line, which now begins with an asterisk.
- ^O inserts a blank line after the current line.
- ESC 2 ^N moves you two lines down, past the new blank line to the next line with text, leaving you ready to repeat the macro.
- ^X) ends the macro definition

The ^X) request accepts a numeric argument; if given one, it executes the macro as well as ends its definition. The number of times it executes it depends on the numeric argument. See "Executing a Macro" below for details.

EXECUTING A MACRO

^XE

After a macro has been defined, ^XE, execute-last-editor-macro, executes it once. The macro to be executed with this request must be the latest one written with ^X(and ^X). With a numeric argument, ^XE executes the macro the number of times specified by the argument, according to the following:

Numeric Argument	Executions
0 (i.e., ESC 0 ^XE)	Repeats execution, with a pause after each, as long as you type a space during the pause. Typing a carriage return or ^G stops the repetition.
1 - 9999 (e.g., ESC 6 ^XE)	Repeats execution the specified number of times.
>9999	Repeats execution until an error occurs.

The ^X) request, which ends a macro definition, can be given a numeric argument, too, specified right in the macro. The numeric arguments are interpreted the same as those for ^XE, above. After you type the ^X), execution starts automatically. Try executing the macro you typed that inserts paragraph "breaker lines."

MID-MACRO QUERY

^XQ

What if you want to execute a macro selectively, i.e., have it affect some cases and ignore others, or stop executing altogether? While you are creating a macro, you can write the ^XQ request, macro-query, into it so that Emacs will stop in mid-execution, letting you control how or whether it continues. You simply type a ^XQ into your macro at the point where you want the pauses to occur. A message appears in the minibuffer:

Inserting query at this point.

During execution of the macro, Emacs performs as much of the macro as precedes the query, then stops and asks you, in the minibuffer, "ok?". You must then type one of the following responses:

- A space (hit the space bar) continues execution of the macro; the requests following are performed.
- A carriage return starts the macro over from its beginning, without performing the requests following the query.
- A ^G stops the macro altogether.

Using ^XQ, then, type in this macro (spaces should not be typed, or they will be inserted into your buffer; they are included in the examples only for readability).

```
^X( ESC] ^XQ ^M **------------** ^X)
```

This macro goes to the end of the current paragraph (by finding a blank line or indented line), queries you, and then opens a new blank line (^M is equivalent to CR and is used here in the definition of the macro to avoid confusion with the CR response to the mid-macro query during execution of the query) and inserts the asterisks and hyphens. During its execution, you are queried at the end of each paragraph. Typing a space in response to the query opens a blank line and fills it with the supplied characters to form a more visual break between paragraphs. Typing a carriage return (perhaps because the "paragraph end" is not really the end of one of your paragraphs or you do not want the current paragraph to be set off) goes to the end of the next paragraph without adding a blank line and the asterisks and hyphens.

DISPLAYING A MACRO

^X*

The ^X* request, show-last-or-current-macro, provides a local display of the last macro defined with ^X(and ^X). The requests are shown as keystrokes, like this:

```
esc-] ^XQ ^M "**------------**"
```

Character strings within the macro are quoted.

If you give this request a numeric argument, e.g. ^U^X*, the keystrokes and command names are displayed. Try it out to see what this display looks like for the macro you typed in.

SAVING A MACRO

ESC X save-macro

You can save the current macro by assigning a command name to it. At the same time, you can assign a key to the named macro, so that you only need type the assigned key to execute the macro. When you invoke the ESC X save-macro extended request, you are prompted for a command name to assign to the macro, and a key sequence:

```
Macro Name?  paragraph-stars
On what key?
```

If you type ^X9, for example, to the second prompt above, you get:

```
On what key?  ^X (prefix char):  9
```

A null response (typing a carriage return) to the key prompt does not assign paragraph-stars to any key; the ESC X set-key extended request, described below, can be used later to assign a key. Key assignments made with ESC X save-macro and ESC X set-key are only effective in the current buffer. Be careful when setting the key that auto-linefeed is turned off on your terminal, or you will end up with ^J as your key every time.

Once a key has been assigned, typing it invokes the macro to which it is assigned. The key can be given any of the numeric arguments accepted by ^XE.

To check your key assignment, invoke ESC ? with the new key. This display appears:

```
^X9             paragraph-stars

^X9 is a keyboard macro.  Type esc-X show-macro
paragraph-stars to display its definition.
```

Had you assigned one of the keys already used for a Fundamental mode request, you could still issue the Fundamental mode request as an extended request. Simply type an ESC X and the command name to do so, e.g., ESC X put-variable, if you wish to assign ^XX to a macro. Actually, you can issue any standard Emacs request as an extended request, at any time.

Displaying a Saved Macro

ESC X show-macro

The ESC X show-macro extended request displays a macro that you have defined with ^X(and ^X), and then assigned a name to with ESC X save-macro. When invoking this request, you must type the name of the macro as an argument, after typing ESC X show-macro and before typing the carriage return that terminates the prompt. The local display is the same as that for ^X*.

EDITING A MACRO

ESC X edit-macros

The ESC X edit-macros extended request produces a symbolic file of all keyboard macros defined in the current buffer, and places it in a new buffer called "Macro Edit." The keyboard macros can then be written out for later loading, and can be edited, redefined, or compiled into Lisp code. This request is also helpful if you have forgotten what macros are available to you. For information on the other uses of ESC X edit-macros, see Appendix D.

SETTING AND CHANGING KEY BINDINGS

ESC X set-key and ESC X set-permanent-key

You can assign your own key bindings to requests and named macros with the ESC X set-key and ESC X set-permanent-key extended requests. The former assigns the key only in the current buffer, and the latter assigns the key in all buffers. Aside from this, both requests work in the same way.

After typing in either request, you must provide two arguments before ending the prompt with a carriage return. The first argument is the key name; the second is the command name. These requests make the key, assigned by the key name, execute the request, specified by the command name (either in the current or all buffers, depending on which key-setting request is used).

The command name can be any standard Emacs command name, or the command name that you have assigned to a macro via ESC X save-macro.

The key name can be the typed representation of a key, i.e., you must type out the letters or characters that represent any of the special keys, rather than typing the special key. For example, in a key name, you can type the word "control" or the caret character, but not the control key. Likewise, you can type the letters "meta" or "m", but not the meta key (found, for example, on Massachusetts Institute of Technology Knight TV consoles or Stanford Artificial Intelligence Laboratory consoles). Key names can have any of the following forms (angle brackets are included to delineate components that are not literal):

◙	<syllable>	e.g., ^f (caret f) - a single keystroke
◙	esc-<syllable> escape-<syllable> ^[<syllable> ^[-<syllable> meta-<syllable> m-<syllable>	e.g.,esc-f or ESC-f escape-f ^[f (^[and ^[- are valid representations of esc) ^[-f meta-f m-f (2 keystrokes, where the first is the ESC, or a single keystroke where the meta key is held down)

- <syllable><syllable> e.g., ^X^C
  <syllable>-<syllable> e.g. ^Z-^@ (2 keystrokes, where the first is some prefix character - not the ESC key)

The following key names cannot be changed:

esc-<digit>
esc-- (esc-<minus sign>)
esc-+ (esc-<plus sign>)

A syllable can be:

- ^ (caret character) (this represents the actual caret key, as in esc-^)
- ^<character> (where character can be any of the upper or lowercase alphabetics, which are equivalent, or [,], \, ^, _, or @. These are interpreted as control characters)

- c-<character>

  ctl-<character>
  control-<character>

  (character is the same as above; these are also control characters)

- esc
  CR
  \177
  TAB
  space or sp

  (where these are all letters or characters representing special keys, and upper and lowercase letters are equivalent)

- <character>

  (where, if the first syllable is ^ in the <syllable><syllable> forms, the character is restricted as above. If the key name is simply this one syllable, uppercase assigns a different key from lowercase. This is the only syllable type where upper and lowercases are not equivalent.

Always remember that you do not actually type any special keys, but only their representations. In general, if you are unsure if a key name is acceptable or not, use a form like one of those found in this manual. Some choices that are valid are not suitable, e.g., a space or alphabetic alone, since you probably need those characters when entering text.

EXAMPLES OF ACCEPTABLE FORMS OF KEY NAMES

^X	^x
^Xq	^P
ESC-ESC	esc-^f
c-p	CR
control-p	^X-^F
\177	^X-CR
#	meta-f
X	TAB
sp	SPACE

See the wall chart (made via ESC X make-wall-chart) for more examples of valid key names.

Two examples of setting keys are given below:

ESC X set-key ^T quit-the-editor

allows you to quit the editor, from the current buffer, in one keystroke. After setting this key (by typing in a caret and a t), you quit the editor by typing, in the usual way, a ^T.

ESC X set-permanent-key ^X9 paragraph-stars

locates the end of the current paragraph and inserts a blank line and a breaker line of asterisks and dashes whenever you type a ^X9 in any buffer.

The user should be aware of the following equivalences between characters. On any ASCII terminal, the linefeed key generates a ^J, etc:

```
\010  =   ^H = backspace
\011  =   ^I = TAB
\012  =   ^J = linefeed
\013  =   ^K = vertical tab
\014  =   ^L = formfeed
\015  =   ^M = carriage return
```

SECTION 16

MULTIPLE WINDOWS AND THE BUFFER EDITOR

Emacs allows the editing of many documents at once; documents are read into buffers, and Emacs can have as many buffers as needed. A unique feature of Emacs, however, is that of displaying multiple documents on the screen at once. This is very useful when writing one document while reading another, such as responding to mail, correcting programming errors while reading compiler diagnostics, merging or comparing programs, and so forth.

Multiple documents (more precisely, multiple buffers) can be displayed by dividing the screen into windows. Normally, the screen consists of one window, called the main window. When you switch buffers (for instance, with the ^XB (select-buffer) request, the new buffer is displayed in the main window, and the mode line is updated to indicate which buffer is on display in that window.

You create new windows with the window-creating requests, described below, and can display any buffer in any of these windows. The windows are divided from each other on the screen by lines of dashes (--------). The cursor is always in some particular window; that window is called the selected window. You select a window i.e., move the cursor into it, with the window-selecting requests, or the window editor. When a window is selected, the buffer on display in that window is being edited, and all the Emacs requests can be used on text it contains.

An optional feature, called pop-up-windows, in which windows are created as needed by various Emacs requests, and windows are destroyed as space is needed, is described in Appendix H.

Before the descriptions of the common window-manipulation requests, here are some terms used in talking about windows:

- buffer - a body of text in Emacs identified by a buffer name. You are already familiar with this concept.

- window - an area of the screen delimited by "boundary lines", i.e., lines of (-----), the top of the screen, or the mode line. A window is said to be displaying a buffer if the text of that buffer can be seen in that window.

- "on display in" - a buffer is on display in a given window if the text of that buffer can be seen in that window.

- topline - the boundary line on the top of a window. The uppermost window has no topline.

- bottomline - the boundary line on the bottom of a window. The bottom-most window has no bottomline.

- selected window - the window in which the cursor now appears (when not in the minibuffer).

- current buffer - the buffer on display in the selected window. The mode line gives the name of the current buffer.

- LRU window - the least recently used window, i.e., the window which has been the selected window least recently.

- previous window - the next-most recently used window other than the selected window itself. The selected window is always the most recently used.

There are several basic techniques for manipulating windows. The simple keyboard requests ^X0, ^X0, ^X1, ^X2, ^X3, and ^X4 can be used to create, destroy, and select windows. At low speed, this may be the only convenient way to edit several "pages" at once. Alternatively, the "window editor" can be used. The window editor, invoked by ^Z^W, puts up a display of the numbers, positions, sizes, and contents of all extant windows, and allows destruction, selection, and size-adjustment of windows by positioning to the line describing the window to be dealt with and issuing requests.

ADDING WINDOWS

^X2

The ^X2 request, create-new-window-and-go-there, creates a new window at the bottom of the screen, and selects that window. The window sizes adjust so that all windows are the same size. The buffer placed on display in the new window is one whose name is constructed as "Window ## Default" (where ## is the window number, the top one being window 1). Thus, if you issue ^X2 when you have only one window, the second window displays a buffer named "Window 2 Default." No arguments are accepted by ^X2.

^X3

The ^X3 request, create-new-window-and-stay-here, creates a new window at the bottom of the screen, but keeps the currently selected window selected. The name of the buffer placed in that window is constructed as described above. The new window becomes the LRU window, so a ^X4 request selects the window created by ^X3 if it is not used before the ^X4 is issued (see ^X4, below).

REMOVING WINDOWS

^X1

The ^X1 request, expand-window-to-whole-screen, removes all windows except the currently selected window, which then grows to occupy the whole screen. Removing a window does not mean getting rid of the text or buffer that is on display in that window; it just means taking the window off the screen.

^XØ

The ^XØ request (control x zero), remove-window, removes the selected window from the screen, giving the space it occupied to the windows that were on either side of it. The previous window becomes the new selected window. With a positive numeric argument, removes the window specified, where the topmost window is 1.

SELECTING A WINDOW

^XO

The ^XO request (control x "oh"), select-other-window, selects the previous window, which is the window you were last in before you were in this window. With only two windows, ^XO selects the "other" window. The cursor appears at the point where it last was in this window. Note that the window in which you issue the ^XO now becomes the previous window, so successive ^XOs switch windows back and forth. Selecting a window, of course, may potentially (and usually does) switch buffers, too. The mode line always tells you what buffer is current; the cursor tells you what window is selected.

^X4

The ^X4 request, select-another-window, selects the LRU window. With a positive numeric argument, e.g., ESC 3 ^X4, it selects the specified window (window 3, in this case). The topmost window is window 1.

A good use for this request with no numeric argument is to select a window you have not been using much (hence, its LRU status) whose contents you can therefore afford to overwrite for some other purpose, such as ^XB or ^X^F. Selecting the LRU window makes it the most recently used (i.e., selected) window. So some other window is now LRU, and another ^X4 selects that window. Thus, successive ^X4's (or ^X4 ^C ^C ...) cycles through all windows on the screen.

EDITING WITH MULTIPLE WINDOWS

The standard Emacs requests work in their usual fashion when you are editing with more than one window. You can edit the material in one window, switch to another (via ^XO, ^X4, and some requests provided by the window and buffer editors described below), and edit the material there. You can, however, only edit in one window at a time. While you are editing the material in that window, you frequently refer to the display(s) in the other window(s). If you need to see more of the buffers displayed in the other windows, switching to them simply to display another set of lines, and then switching back, is very inconvenient. This is especially true if you are using several windows, so that each portion displayed is relatively small. The ESC ^V request solves this problem.

ESC ^V

If you want to "turn the pages" in the windows that are not selected, you can do so without switching to them. The ESC ^V request, page-other-window, is only valid when more than one window exists. Without a numeric argument, ESC ^V displays the next windowful (as with a ^V) of the "other" window. The other window is the unselected window when only two windows exist; when more than two exist, the other window is the next most recently used window. With a positive numeric argument, e.g., ^U ESC ^V, this request goes forward the specified number of screensful (4 in this case) and displays it. With a negative numeric argument, it pages the other window backward the specified number of screensful.

The ESC ^V request is, of course, most useful if you need to refer frequently only to one other window (if you are, for example, responding to mail). If you have to update several windows, however, you must switch windows. Use ^X4 to go to whichever window(s) you want, then use ^V or ESC V as usual, and return to the original windows with ^X4.

One word of warning with multiple-window editing: if you display the same buffer in more than one window at once, Emacs becomes slower, less efficient, and substantially more expensive to use while the buffer is so displayed. Avoid entering text into such a buffer if any of these issues are a concern.

DEDICATED BUFFERS

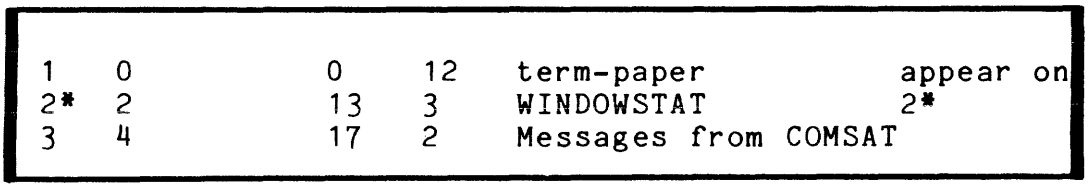
Several Emacs requests always switch you into a new buffer dedicated to their exclusive use, and leave you in that buffer. The window editor (^Z^W), the comout-command request (^X^E) described in Section 17, which allows you to execute a Multics command and displays the output of that command in a "file_output" buffer, and the Emacs mail requests (^XM and ^XR) described in Appendix B, are all examples of requests requiring their own buffers. These requests are often issued while you are in the midst of editing, and you generally do not want them to obliterate your work from the screen. Thus, when you use any of them, you probably want at least one extra window available. The ^X2 or ^X3 request is a good way to get the extra window.

Some dedicated-buffer requests select the window that already contains their appropriate buffer (e.g., ^X^E selects "file_output" if it is on display in any window). If the appropriate buffer does not exist, they either select the LRU window, replacing that window's previous contents with their display, or they use the current window for their display. The choice of LRU or current window depends on the request. The ^X^E, ^XM, and ^XR requests choose the LRU window, but ^Z^W chooses the current window. To make a window available without endangering any windows already in use, use ^X2 for those dedicated-buffer requests utilizing the current window, and ^X3 for those utilizing the LRU window. This process of selecting the window with the appropriate buffer is called find-buffer-in-window (and is available to programmers writing extensions. See the Extension Writers' Guide).

THE WINDOW EDITOR

^Z^W

The window editor provides an interactive way to manipulate windows, allowing you to reorganize the screen conveniently. The window editor puts a formatted display in a dedicated buffer. The display appears on the screen in the selected window (the window where the cursor is currently sitting). The window editor buffer is named WINDOWSTAT. If you want WINDOWSTAT to appear in a window other than the current one, issue ^Z^W with a numeric argument, e.g., ^U^Z^W. If WINDOWSTAT is already displayed in another window, the new display appears there; otherwise, the new display appears in the LRU window.



Each line relates to the contents of one window on the screen at the time the window editor was entered. There are as many lines as there are windows. You cannot use standard Emacs requests to change the contents of the display; it is read-only. You can, however, use standard Emacs requests (e.g., search requests, etc.) to position the cursor in it. The cursor, when in the window editor's buffer, is always on some line of the buffer (the buffer may be larger than the window it is in, like most Emacs buffers). That line relates to one of the windows; the window's window-number from the top of the screen is the first number on that line. The window designated by the line on which the cursor is will be called the "designated window" (do not confuse this with the selected window, which is the window containing the window editor's display while you are working in the window editor).

The window number followed by a star shows which window was the selected window at the time the window editor was entered. The next number on each line is called the internal window number, and is usually not of interest. The remaining two numbers on each line are the position and size of each window. The position is the screen's line number at which the window starts (the screen's top line is 0). The size is the number of lines in the window.

Following the position and size is the buffer name of the buffer currently on display in that window. Following the buffer name are the first ten characters of the point line in that window. The point line is the line in the window that the cursor goes to if that window is selected (generally the line where the cursor last was in that window).

Window Editor Requests

You operate the window editor by invoking it, positioning to some line, thus designating some window, and issuing window editor requests to affect that window. The following requests are recognized (note that they are printing characters, instead of control characters, for ease of typing, since you cannot enter text into a read-only buffer).

g

Goes to (selects) the designated window, leaving the window editor, and moving the cursor to this window.

f

Goes to the designated window (same as g, for compatibility with the buffer editor).

k

Kills (removes) the designated window from the screen. This is done immediately, and the buffer editor display is updated to reflect the new screen layout. The space occupied by this window is distributed among its neighbors.

d

Kills the designated window (same as k, for compatibility with the directory editor. See ^XD in Section 17 for information on the directory editor).

^ (caret)
Moves the the topline of the designated window up one line, increasing its size, and deducting one line from its neighbor above. With a numeric argument, moves up that many lines instead of one. The buffer editor display is updated to reflect the new screen layout. (The shapes of the ^, V, A, and U requests suggest their function.)

v
Moves the bottomline of the designated window down, same rules and features as ^.

a
Moves the topline of the designated window down, same rules and features as ^.

u
Moves the bottomline of the designated window up, same rules and features as ^.

The following requests do not deal with the designated window, and can be issued at any time in the window editor:

n
Goes to the next line of the window editor display. You could use ^N as always, but n is easier in a read-only buffer. If on the last line, goes to the first.

p
Goes to the previous line. If on the first line, goes to the last.

b
Exits the window editor by entering the buffer editor in the window now occupied by the window editor's display.

c
Creates a new window (like ^X3), and leaves you in the window editor's display. The display is updated to reflect the new state of the screen, with the new window as designated window.

3
Creates a new window (same as c, for mnemonic ease with ^X3).

LEAVING THE WINDOW EDITOR

The window editor is usually exited by selecting some other window with the g request; indeed, you may often enter the window editor for no other reason than to do this. You can also exit via the b request to the buffer editor, or just issue a ^XB or ^X^F to go elsewhere. Window editor windows left on the screen after you exit the window editor are not updated dynamically by Emacs when windows and buffers are changed around.

THE BUFFER EDITOR

The buffer editor provides a facility for deleting, examining, and selecting buffers, similar to the window editor. The buffer editor creates a read-only display in a dedicated buffer. The display appears on your screen in the selected window. As with the window editor, several buffer editor requests, invoked by typing selected printing characters, allow you to manipulate the buffer designated by the cursor's position in the display.

^Z^B

The buffer editor is entered via ^Z^B. It puts its display, a buffer named BUFED, in the window in which it was invoked. If you do not want to overwrite the selected window, issue ^Z^B with a numeric argument, e.g., ^U^Z^B. The buffer editor's display then appears in the window already displaying it, if one exists; otherwise, it appears in the LRU window.

Each line of the buffer editor's display contains a buffer's name, a pathname if the buffer has a pathname associated with it, and, possibly, flags. The flags appear at the left margin, and they are:

*	This buffer is modified (needs writing out).
>	This buffer was current when the buffer editor was entered.
X	This buffer is marked for deletion by the buffer editor.

There may be more buffers (i.e., lines in the buffer editor's display) than are on display in the window in which this display appears; like any other Emacs buffer on display, ^V, ESC V, or any other standard Emacs request can be used to position the cursor in it.

Buffers can be killed (as ^XK does) with the buffer editor k or d request. Buffers so killed are not actually destroyed until the buffer editor is exited via a g, w, f, q (or ^X^Q) request, at which time you are asked if you really want to delete them (they are listed in a local display).

Buffer Editor Requests

The following requests are avaliable in the buffer editor (the uppercase equivalent of each request is also acceptable, e.g., n or N goes to next line):

n

Goes to the next line, or the first line if now on the last.

p

Goes to the previous line, or the last line if now on the first line.

k

Marks the designated buffer for deletion when the buffer editor is exited and moves to the next line.

d

Marks the designated buffer for deletion (same as k, for compatibility with the directory editor).

s

Writes the designated buffer out to its default pathname (it must have one). This marks it as unmodified.

u

Undoes the effects of k or d on the designated buffer, i.e., unmarks it. The X flag is removed from the display, and the cursor is positioned to the next line.

e

Examines the designated buffer. In one-window mode, you should just go there. With two or more windows, Emacs selects a window via find-buffer-in-window for the designated buffer, putting it on display if not already on display. A message is printed in the minibuffer about where (in what window) it appears.

LEAVING THE BUFFER EDITOR

These next requests cause the buffer editor to be exited; you are queried about pending deletions if you have any (uppercase equivalent of each request is also acceptable):

g

Goes to the designated buffer, exiting the buffer editor, replacing its display in the current (selected) window, with this buffer. This is just like doing a ^XB. As a matter of fact, the buffer editor can be used for just seeing what buffers there are and going to one, to save typing. This is especially useful for buffers with long and complex names, like "Messages from Brzezinski."

f

Selects a window and displays the designated buffer via find-buffer-in-window. Since the buffer editor makes its own window LRU when it exits, if you "f" a buffer not currently on display in any window, it is the same as going to it via the g request, replacing the buffer editor's window.

However, if the designated buffer is on the screen somewhere, the cursor simply moves into that window (and thus, into that buffer). If you use multiple windows, you will find that you use "f" all the time, and rarely use "g".

w

Exits the buffer editor by entering the window editor in the window now occupied by the buffer editor's display.

q

Exits the buffer editor and enters, in the current window, the buffer from which it was invoked by ^Z^B. If you invoked the buffer editor with a numeric argument, e.g., ^U^Z^B, this exits the buffer editor and enters, in the appropriate window, the buffer from which you invoked it.

^X^Q

Exits the buffer editor in the same manner as with "q."

SECTION 17

SUMMARY OF EMACS FUNDAMENTAL MODE REQUESTS

The following requests have been grouped according to the functions they perform. They are the requests available in Fundamental mode, which is the default mode entered in new buffers. Some may appear more than once if they serve more than one purpose. To invoke the extended requests, type an ESC X followed by the command name. If the extended request requires an argument, type it after the command name (with a space separating the two). Then, with or without argument(s), type a carriage return. The following is a list of the functions and the requests (by key) documented within each group. After this list, full descriptions of each request in each of the groups are given.

LIST OF EDITING FUNCTIONS AND THE KEYS THAT PERFORM THEM

Movements Forward/Backward

^B	^F	^A	^E
^N	^P	ESC B	ESC F
ESC A	ESC E	ESC [	ESC]
^V	ESC V	ESC <	ESC >
ESC G	ESC ^B	ESC ^F	

Deletion

#	\177	^D	@
^K	ESC #	ESC \177	ESC D
^W	^X#	^X\177	ESC K
^XK	ESC ^W	ESC ^Y	^X^O
^Z;			

Retrievals/Yanks

^Y	ESC Y	ESC ^Y

Marks, Regions, Variables

^@	ESC H	^XH	^X^X
^Z^@	^ZG	^W	ESC W
^X^L	^X^U	^XX	^XG
^X_			
ESC X lvars			

Searches and Substitutions

^R	^S	ESC /	^XW
^XS	ESC %		
ESC X replace			

Files

^X^F	^X^R	^X^S	^X^W
^ZF			

Insertion

^XI	^Z^F

Entry and Exit

^X^C	^XCR	^X^E	^Z^Z
^XD			

Help

ESC ?	^_
ESC X apropos	
ESC X describe	
ESC X make-wall-chart	

Error Recovery

^G	^X^G	^Z^G	ESC ^G

New Lines/Blank Lines

CR	^O	^X^O	ESC ^O

Indentation and White Space

ESC \	ESC M	^X.	^XF
ESC I	ESC CR	ESC ^I	ESC ^
^X^I	^O		

Comments

^X;	ESC ;	^Z;	ESC N
ESC P			
ESC X set-comment-prefix			

Formatting

^X.	^XF	ESC S	ESC Q
ESC X fillon			
ESC X filloff			
ESC X runoff-fill-region			

Literal Character Entry

\	^Q

Special Purpose Keys

^J	^L	^U	ESC
^XESC	ESC X	^Z^L	

Macros

^X(	^X)	^X*	^XE
^XQ			
ESC X save-macro			
ESC X edit-macros			
ESC X show-macro			

Characters (Moving by/Deleting)

^B	^F	#	\177
^D			

Lines (Moving in and by/Deleting)

^A	^E	^N	^P
^X=	ESC G	@	^K

Words

ESC B	ESC F	ESC #	ESC \177
ESC D	ESC C	ESC L	ESC U
ESC _	^Z_	ESC T	

Sentences

ESC A	ESC E	^X#	^X\177
ESC K			

Paragraphs

ESC [	ESC]	ESC H

Screens

^V	ESC V	^L	ESC R
^Z^V			

Buffers

ESC <	ESC >	^XB	^X^B
^XH	^XK	ESC ~	^Z^B

Multiple Windows

^XØ	^X1	^X2	^X3
^X4	^XO	ESC ^V	^Z^W

Mail/Messages

^XM	^XR

ESC X accept-messages
ESC X accept-messages-path

Typing Shortcuts

```
^C          ^T          ^U          ^\
ESC T
ESC SPACE
ESC X setab
ESC X speedtype
ESC X speedtypeoff
```

Programming Modes

```
ESC X alm-mode
ESC X electric-alm-mode
ESC X electric-pl1-mode
ESC X fortran-mode
ESC X fundamental-mode
ESC X lisp-mode
ESC X pl1-mode
ESC X set-compile-options
ESC X set-compiler
ESC X ldebug
ESC X text-mode
```

Printing Terminal Usage

```
^XV         ^X^T        ^O          ^Z^L
```

Extension Writing

```
ESC ESC
ESC X ldebug
ESC X loadfile
ESC X loadlib
```

Additional Optional Settings

```
ESC X opt
ESC X option
ESC X set-minibuffer-size
ESC X reset-minibuffer-size
ESC X set-screen-size
ESC X reset-screen-size
ESC X set-key
ESC X set-permanent-key
ESC X set-search-mode
```

DESCRIPTIONS OF THE REQUESTS

Movements Forward/Backward

^B backward-char
Moves backward one character in the buffer. Tabs and the newline characters at the ends of lines count as one character. Beeps as for ^G at the beginning of the buffer. Repeats with a positive numeric argument; moves forward and repeats with a negative numeric argument.

^F forward-char
Moves forward one character. Tabs and newlines count as one character each. Beeps as for ^G at the end of the buffer. Repeats with a positive numeric argument; moves backward and repeats with a negative numeric argument.

^A go-to-beginning-of-line
Positions to the beginning of the current line of the buffer, i.e., right before the first character.

^E go-to-end-of-line
Positions to the end of the current line, i.e., after the last character and before the newline. On an empty line, this is the same as the beginning of the line.

^N next-line-command
Positions to the next line of the buffer. If on the last line, appends a new empty line to the bottom of the buffer, and positions to the beginning (and end) of it. Successive ^Ns and ^Ps try to maintain the same horizontal position. Repeats with a positive numeric argument; performs ^Ps and repeats with a negative numeric argument.

^P prev-line-command
Moves to previous line of the buffer. Beeps as for ^G if on first line of the buffer. Attempts to maintain the same horizontal position; successive ^P's and ^N's try to maintain the original horizontal position. Repeats with a positive numeric argument; moves to next line and repeats with a negative numeric argument.

ESC B backward-word
Goes backward one word. If in the middle of a word, goes to before the beginning of that word. Skips backward over all white space to get to the next word. Underscores and backspaces count as parts of words. Repeats with a positive numeric argument; goes forward and repeats with a negative numeric argument.

ESC F forward-word
Goes forward one word. If in the middle of a word, moves to the end of the current word. Leaves point immediately after that word. Passes over all punctuation and white space before the word. Underscores and backspaces count as parts of words. Repeats with a positive numeric argument; moves backward and repeats with a negative numeric argument.

ESC A backward-sentence
Goes to the beginning of the current sentence, i.e., just before the first letter. If already at the beginning of a sentence, goes to the beginning of the previous sentence. The beginning of the first word after a blank line always counts as the beginning of a sentence. Repeats with a positive numeric argument; goes forward and repeats with a negative numeric argument.

ESC E forward-sentence
Moves forward to the end of this sentence. If at the end of a sentence, moves forward to the end of the next sentence. Ends of paragraphs are implicitly ends of sentences, whether or not an end-of-sentence punctuation (period, question mark, exclamation point) appears. Repeats with a positive numeric argument; moves backward and repeats with a negative numeric argument.

ESC [beginning-of-paragraph
Moves to the beginning of the current paragraph. If already at the beginning of a paragraph, moves to the beginning of the previous paragraph. The beginning of a paragraph is the beginning of the first line of the paragraph. The definition of paragraphs is controlled by the paragraph-definition-type option: if 1, blank lines separate paragraphs; if 2, an indented line starts a paragraph. The Multics runoff or compose command's control lines count as individual paragraphs. Repeats with a positive numeric argument; moves forward and repeats with a negative numeric argument.

ESC] end-of-paragraph
Moves to the end of the current paragraph. If at the end of a paragraph, moves to the end of the next paragraph. The end of a paragraph is the end of the last line of the paragraph. The definition of paragraphs is controlled by the paragraph-definition-type option: if 1, blank lines separate paragraphs; if 2, an indented line starts a paragraph. The Multics runoff or compose command's control lines count as paragraphs. Repeats with a positive numeric argument; moves backward and repeats with a negative numeric argument.

^V next-screen
Displays next screenful of the current buffer. Leaves cursor at upper left hand corner of screen. With a positive numeric argument, pages forward the specified number of screensful and displays it; with a negative numeric argument, moves backward the specified number of screensful (previous screens) and displays it.

ESC V prev-screen
Displays the previous screen (one back) of this buffer, leaving cursor at the top of it. With a positive numeric argument, moves backward the specified number of screensful and displays it; with a negative numeric argument, moves forward the specified number of screensful and displays it.

ESC < go-to-beginning-of-buffer
Moves to the beginning of the current buffer, i.e., before the first character in the buffer at the top of the document being edited.

ESC > go-to-end-of-buffer
Moves to the end of the current buffer, i.e., after the last character or newline character at the bottom of the document being edited.

ESC G go-to-line-number
Goes to a given line, specified by line number, from the top of the buffer. The positive numeric argument specifies the line number. For instance, ESC 25 ESC G goes to line 25.

ESC ^B balance-parens-backward
Skips backward over one set of balanced parentheses. Searches backward until a set of parentheses is found. Does not handle quoting or any programming language conventions.

ESC ^F balance-parens-forward
Skips forward over one set of balanced parentheses. Searches forward until a set of parentheses is found. Does not handle quoting, or any other programming language conventions.

Deletion

rubout-char
Deletes the previous character (before the cursor, which is usually the last character typed). Note that # deletes the character to the left of the cursor, while ^D deletes the character at the cursor. Repeats with a positive numeric argument, deletes the character at the cursor and repeats with a negative numeric argument.

\177 (delete key) rubout-char
Deletes the previous character (before the cursor, which is usually the last character typed). Note that \177 deletes the character to the left of the cursor, while ^D deletes the character at the cursor. Repeats with a positive numeric argument; deletes the character at the cursor and repeats with a negative numeric argument.

^D delete-char
Deletes the character to the right of the current point. This is the character on which the cursor sits. Moves the rest of the line one to the left, closing up the space. Deleting a newline at the end of a line merges lines. Repeats with a positive numeric argument; deletes the previous character and repeats with a negative numeric argument.

@ kill-to-beginning-of-line
Kills all the text to the left of the cursor on the current line. The killed text is saved on the kill ring, and may be retrieved with ^Y.

^K kill-lines
Kills to end of line; when already at end of line, deletes the newline (merges lines). If on an empty line, deletes it. If given a positive numeric argument, deletes that many lines, starting from the current point on the current line. Successive ^Ks merge killed text on the kill ring.

ESC # rubout-word
Deletes the word to the left of the current point. More specifically, deletes going backward, deleting characters until the beginning of a word. Successive words deleted with ESC # are merged and can be retrieved with one ^Y. Repeats with a positive numeric argument; deletes forward and repeats with a negative numeric argument.

ESC \177 rubout-word
Deletes the word to the left of the current point. More specifically, deletes going backward, deleting characters until the beginning of a word. With a positive numeric argument, deletes the specified number of words. Deletes forward the specified number of words with a negative numeric argument. Successive words deleted with ESC \177 are merged and may be retrieved with one ^Y.

ESC D delete-word
Deletes the word to the right of the current point. More specifically, deletes forward, deleting all white space and punctuation and characters until the end of the next word. Repeats with a positive numeric argument; deletes backward and repeats with a negative numeric argument.

^W wipe-region
Wipes (kills) all text between cursor and the mark. Does not participate in kill merging, since ^Ws cannot be successive. The killed text is saved on the kill ring and can be retrieved with ^Y.

^X# kill-backward-sentence
Kills backward to the beginning of this sentence; kills as much of the sentence as thus far typed. Successive kills via ^X# and other reverse-killing commands (e.g., ESC #) merge, and may be retrieved with one ^Y. Repeats with a positive numeric argument; kills forward and repeats with a negative numeric argument.

^X\177 kill-backward-sentence
Kills backward to the beginning of this sentence; kills as much of the sentence as thus far typed. Successive kills via ^X\177 and other reverse-killing requests (e.g., ESC #) merge, and may be retrieved with one ^Y. Repeats with a positive numeric argument; kills forward and repeats with a negative numeric argument.

ESC K kill-to-end-of-sentence
Deletes text going forward from the cursor to the end of the current sentence. If at the end of a sentence, deletes forward to the end of the next sentence. Sentences and other text killed consecutively in this fashion are merged, and may be retrieved with a single ^Y. With a positive numeric argument, kills the specified number of sentences. Kills backward the specified number of sentences with a negative numeric argument.

^XK kill-buffer
Kills (destroys) a buffer. Prompts for the buffer's name, terminated by CR. A response of CR kills the current buffer. Buffers can be killed to conserve storage, to prevent their appearance in buffer listings, or to prevent being queried when quitting with ^X^C.

ESC ^W merge-last-kills-with-next
Causes the next kill-type request (e.g., ^K, ESC D), which must follow immediately, to merge what it kills with the last saved kill on the kill ring, in the same direction as the next request kills. For instance, ^A ^K ^K ^N ^N ESC ^W ^K ^K catenates two disjoint lines on the kill ring.

ESC ^Y yank-minibuf
Yanks back the last contents of the minibuffer, without a prompt string. The mark is set in the minibuffer, so ^X^X can be used to position around it, and ^W to delete it. The real mark in the main buffer is not destroyed.

^X^O delete-blank-lines
Deletes blank lines around cursor: gets rid of vertical white space. If issued on a blank line, leaves one blank line; otherwise, deletes all blank lines after this line's end. See ^O.

^Z; kill-comment
Removes the comment and the white space preceding it from the current line. The deleted text is saved on the kill ring, accessible to ^Y. The text is saved in such a way that following ^Ks and other forward-killing requests merge properly with the deleted text.

Retrievals/Yanks

^Y yank
Yanks (retrieves) killed text into place at cursor. Retrieves last killed word, line, or region. With a positive numeric argument, goes that many killings down the 10-position kill ring. Leaves the mark at the front of the retrieved text, and the point at the end.

ESC Y wipe-this-and-yank-previous
Deletes the text between the point and the mark without saving it, rotates the kill ring one position (slot 2 text now occupies slot 1), and retrieves the text just moved to the first slot of the kill ring.

ESC ^Y yank-minibuf
Yanks back the last contents of the minibuffer, without a prompt string. The mark is set in the minibuffer, so ^X^X can be used to position around it, and ^W to delete it. The real mark in the main buffer is not destroyed.

Marks, Regions, Variables

^@ set-or-pop-the-mark
With no argument, sets the mark in this buffer where the cursor is now, and leaves it there. The current value of the mark, if any, and if different from the current point, is pushed on to the mark ring. The mark relocates to the nearest point if the text around it is deleted. See ^X^X to verify where the mark is. With a positive numeric argument, e.g., ^U ^@, pops the previous mark off the mark ring, and positions to it. Successive ^U ^@s "try" all marks on the mark ring.

ESC H mark-paragraph
Puts the mark at the beginning of the current paragraph; puts the cursor at the end of the current paragraph. See ESC [for a definition of paragraphs.

^XH mark-whole-buffer
Puts the mark at the end of the buffer and the cursor at the beginning. This "marks" the whole buffer, so that ^W deletes it, etc. The linefeed at the end of the buffer is not in the marked region, but ^XH ^W ^XB ... ^Y effectively moves a whole buffer.

^X^X exchange-point-and-mark
Exchanges the cursor and the mark, to verify their positions before typing ^W or similar requests. Puts the cursor where the mark is and vice versa. Typing ^X^X ^X^X quickly verifies the extent of the point/mark region visually and returns the cursor and mark to their original positions. Use ^@ with a positive numeric argument (e.g., ^U^@) to visit older settings of the mark in this buffer.

^Z^@ set-named-mark
Prompts for a name to be associated with a mark, and sets that named mark to be where the cursor is now. Named marks are valid only in the buffer in which they were created. Use ^ZG to go to a named mark.

^ZG go-to-named-mark
Prompts for the name of a named mark, and moves the cursor to the point where that mark was saved. Use ^Z^@ to set a named mark.

^W wipe-region
Wipes (kills) all text between cursor and the mark. Does not participate in kill merging, since ^Ws cannot be successive. The killed text is saved on the kill ring and can be retrieved with ^Y.

ESC W copy-region
Copies the text between the cursor and the mark on to the top of the kill ring. This means that the next ^Y copies the text now between the cursor and the mark to wherever the cursor is when the ^Y is issued.

^X^L lower-case-region
Lowercases all letters between point and the mark.

^X^U
Uppercases all letters between point and the mark.

^XX put-variable
Stores away point/mark region to a variable, whose name is prompted for, terminated by CR. Use ^XG to retrieve this value, and ESC X lvars to list such variables.

^XG get-variable
Gets back a variable stored by ^XX. The name of the variable is prompted for; the cursor is put after it, and the mark before it.

^X_ underline-region
Underlines the entire region. When given any numeric argument, ^X_ removes underlining from the entire region. White space within the region is handled if you set the ESC X opt underline-whitespace option to on.

ESC X lvars
Displays the names and lengths of all variables saved by ^XX. Type ^J to resume, or just continue editing. See ^XX and ^XG.

Searches and Substitutions

^R reverse-string-search
Reverse searches. Leaves cursor positioned before matching string; does not move cursor if not found. Prompts for search string in minibuffer, which must be ended with CR.

^S string-search
Searches for a character string, from current point in buffer to end. Prompts for search string in minibuffer, and leaves point, if search succeeds, after the matched string. End search string with CR. Typing ^S CR reuses last search string. If search fails, point does not move.

ESC / regexp-search-command
Searches forward for a regular expression that is prompted for and terminated by CR. A regular expression is a character string in which the following special characters can be included:

*
matches any number, including none, of the character preceding it.

.
matches any character (.* matches everything).

^
represents an imaginary character preceding the first character on a line (this character is the caret, not the representation for the control key). It must be the first character in a regular expression; its use allows you to search for the line beginning with the next characters.

$
represents an imaginary character following the last character on a line. It must be the last character in a regular expression; its use allows you to search for a line ending with the preceding characters.

To include any of these special characters in the regular expression without their special meaning, you must precede each occurrence of them with \c.

Searches forward from cursor, and can find many occurrences of the regular expression on one line. Leaves the cursor and the mark around the string it finds, so that:

```
s/(fo.*)/(a b & )/                    (qedx)
```

is equivalent to:

```
ESC / (fo.*)CR ^W (a b ^Y )           (Emacs)
```

^XW multi-word-search
Searches for words. Prompts for one or more words, terminated by CR. (This is a search string; typing just a CR in response to the prompt reuses the last search string). Searches from current point to the buffer's end for those words appearing in order, regardless of case of letters, underlining, intervening punctuation, white space, or line breaks. Finds whole words, not parts of words. A partial word ending with * in the search string is matched by any word beginning with the letters provided. With a positive numeric argument, e.g., ^U^XW, goes to beginning of buffer before searching.

^XS global-print-command
Prints all lines containing a given string. Prompts for the string , terminated by CR. With a positive numeric argument, e.g., ^U^XS, takes a regular expression, i.e., the search string can include the special characters *, ., $, and ^ (not the control key, but the caret symbol). See ESC / for the meanings of these special characters in a regular expression. Type ^J or continue editing to restore buffer display.

ESC % query-replace
Interactively replaces all occurrences of one string with another. The request prompts for both strings in the minibuffer, terminated by CR, and then searches forward for each occurrence of the first string. It positions the cursor immediately after this string and waits for one of the following responses (type the appropriate keys):

space
replaces this particular occurrence of the first string with the second. Then searches for the next occurrence of the first string, updates the screen, and waits for a response again.

CR
leaves this occurrence of the first string alone and searches for the next occurrence of the first string.

. (period)
replaces this occurrence of the first string with the second and then terminates the query replace.

^G
terminates the query replace without modifying this occurrence of the first string.

ESC
same as ^G.

!
replaces all occurrences of the first string from the current point to the end, without querying again.

, (comma)
replaces this occurrence of the first string with the second, immediately updating the screen. Then searches for the next occurrence of the first string and waits for a response again.

^L
redisplays the screen.

^_
displays a description of the allowable responses (i.e., prints this list).

?
same as ^_.

ESC X replace
Globally replaces one string with another, from the current point to the end of the buffer. Prompts for two strings, terminated by CR. If the first string is not found, ESC X replace does not move the cursor. Use ESC % if you want to be queried before each replacement occurs.

Files

^X^F find-file
Reads in a file. Prompts for a file's pathname, terminated by CR. This request attempts to switch to a buffer (see ^XB) that contains the specified file, if the file is already in a buffer.

If no such buffer exists, ^X^F reads the file into the buffer whose name is the first component of the entry portion of the filename, and sets the default file of this buffer to the file just read. If the find-file-set-modes option is on, ^X^F sets the major mode of the buffer according to the last component of the entry portion of the filename. For example, for the filename ">ldd>include>sst.incl.pl1", the buffer chosen is "sst" and the major mode is "PL/I." In cases where buffer name duplications could arise (e.g., reading sst.incl.pl1 and then sst.list), ^X^F adds the second component to the subsequent buffer names.

If the file is in more than one buffer, those buffers are listed as if by ^X^B. You are then prompted for the name of the buffer you wish to use, terminated by CR. If the buffer specified is one of those listed, ^X^F switches to it. If a new buffer name is specified, ^X^F reads the file into that buffer as described above. A blank response for the buffer name uses the original buffer named by the first component of the entry portion of the filename. ^X^F accepts archive component pathnames. An archive component pathname is a (relative) pathname of an archive file followed by two colons (::) and a component name (archive::component). ^X^F also accepts starnames (including stars in the archive and component names). ^X^F ignores directories that match the starname. When more than one star match is found, a ^X^F is done for each file/archive component selected. For example, to read in Additions.list:

```
^X^F Add*.*
```

For example, to edit a system source file:

```
^X^F >ldd>h>s>bound_p*.**::page_fault.alm
```

To edit a subset of segments in the same archive and read each file into its appropriate buffer:

```
^X^F >ldd>h>s>bound_p*.**::pc_*.*
```

To read all of an archive into its appropriate buffers:

```
^X^F my_source_archive::**
```

If a buffer contains the requested file and the file being read in has been modified since last read or saved, a local display informs you that the buffer contains an old version of the file, and the minibuffer prompts you to select a choice of actions, after prompting you to select the appropriate buffer, if necessary. See the description of ^X^F in Section 5 for more information on the Emacs query and choices of action.

^X^R read-file
Reads in a file. Prompts for a file's pathname, terminated by CR. Accepts archive component pathnames; also accepts starnames, but no more than one file or archive component can match the starname. Reads that file into the current buffer, destroying anything which was in the buffer, and sets this buffer's default file to the file read. The cursor is left at the first position of the first line of the file read. If a blank response is given for the filename, the buffer's default file is read. The default file is set by ^X^R, ^X^F, and ^X^W. ^X^R is useful for starting again after big mistakes.

If the buffer was modified since last read or written, Emacs queries whether to proceed with read-file. Turning the read-file-force option on (it is off by default) or supplying a numeric argument to the read request (e.g., ^U^X^R) disables this protection feature.

^X^S save-same-file
Writes out the buffer contents to the same file last read in or written out, i.e., writes the buffer to its default file. This request is equivalent to ^X^WCR. Files cannot be written out to an archive component pathname. When the ESC X opt check-newline option is on, this request queries you if you are trying to write out a file that does not end in a newline (this occurs only if you have set the ESC X opt add-newline option off, however).

If the file being saved has been modified since the buffer was last read or written, Emacs queries whether to proceed with the save. See the description of ^X^S in Section 5 for more information on this condition.

^X^W write-file
Writes the current buffer out to a file, whose pathname is prompted for in the minibuffer, terminated by CR. If a blank or null response is given, writes it out to this buffer's default file. The file specified becomes the buffer's default file. Files cannot be written out to an archive component pathname. When the ESC X opt check-newline option is on, this request queries you if you are trying to write out a file that does not end in a newline (this occurs only if you have set the ESC X opt add-newline option off, however).

If the response to the pathname prompt is blank (i.e., same as save-same-file) and the file being written has been modified since the buffer was last read or written, Emacs queries whether to proceed with the save request. If pathname is specified and a file of that name already exists, Emacs queries whether to overwrite the existing file. Both of these conditions are further described under ^X^W in Section 5.

^ZF object-mode-find-file
Enters the object-mode major mode, which is used for editing files in octal. Prompts for the name of the file, then displays the file in the form of four words of storage per line. Each line in the buffer contains three items: 1) the word offset for the first word of storage displayed on that line, 2) the ASCII octal representation of the next four words of the storage in the file, and 3) the printing characters of those four words of storage. Edits by overwriting existing characters only; no editing is allowed for white space. Both octal words and printing characters can be edited. If you edit the octal representation, Emacs adjusts its corresponding printing character representation, and if you edit the printing character, its octal representation is adjusted.

Insertion

^XI insert-file
Inserts a file into the current buffer. Prompts for a file's pathname, terminated by CR. Accepts archive component pathnames; also accepts starnames, but no more than one file or archive component can match the starname. Reads that file into the current buffer without destroying the previous contents of the buffer. The file is read in to the left of the cursor and the cursor is left after the contents of the file just read. The mark is left at the beginning of the inserted text. The default file for the buffer is not changed. (See ^X^S.)

^Z^F get-filename
Inserts the pathname (as seen in the buffer's mode line) of the current buffer at the cursor. With a numeric argument, inserts only the entryname of the pathname; if the pathname is an archive pathname, inserts only the component name. This request is most useful for getting a file's name into the minibuffer.

Entry and Exit

^X^C quit-the-editor
quit
Exxits the editor. If modified buffers exist, they are listed as if by ^X^B; ^X^C then asks you if you really want to exit the editor.

^XCR eval-multics-command-line
Prompts for a Multics command line, terminated with CR, and executes it. Multics commands that produce output may well ruin your screen; if this occurs, use ^L. If you expect output, use ^X^E instead of ^XCR. A ^X^M is equivalent to ^XCR.

^X^E comout-command
Executes a Multics command line (prompted for, end with CR), and displays the command line's output in buffer "file_output." If buffer "file_output" is already on display in a window, the cursor moves to that window, and "file_output" stays there.

^Z^Z signalquit
Signals QUIT to Multics and clears the screen. Restores the tty modes suitable for Multics command usage before doing so, and after you type start, it restores the appropriate tty modes for the Emacs environment.

^XD edit-dir

Enters the directory editor, editing the working directory. With a positive numeric argument, e.g., ^U^XD, prompts for some other directory name. The pathname of the directory being edited is displayed on the path line. Position to a line with some segment's name on it, and the following requests (keys) can be used (lowercase is acceptable):

D

Deletes this segment when directory editor is exited.

U

Undeletes, i.e., cancels previous D on this line.

E

Examines (i.e., takes a look at) this segment, in a separate buffer. Use ^X^Q to get back, and the examine buffer disappears automatically.

Q

Quits the directory editor. A list of files is shown, and you are queried if you want to delete them or not. To exit without any action, use ^XB.

R

Renames a segment (modify access on the directory is required). It prompts for the new name.

N

Same as ^N.

P

Same as ^P.

Help

ESC ? describe-key

Displays the documentation for a given key sequence. For example, to find out what a ^D does, type ESC ? and, when prompted, a ^D. With a positive numeric argument, e.g., ^U ESC ?, displays in the minibuffer just the command name to which the key is currently connected.

^_ help-on-tap

Gets help/documentation at any time. The current repertoire is:

^_H

Shows where to get more help.

^_?

Displays the current repertoire of ^_.

^_L

Displays the last 50 characters typed in.

^_^G

Does a ^G as usual.

^_A

Works like the ESC X apropos extended request.

^_C

Works like ESC ?, describe-key.

^_D
Works like the ESC X describe extended request.

ESC X apropos <string>
Lists all requests and extended requests that contain a given string in their command names, and tells what, if any, keys invoke them in the current buffer. For instance,

ESC X apropos forw

lists forward-word, forward-char, etc. This is the most common way to find a request that does something you are looking for.

ESC X describe <extended-request>
Displays the documentation for an extended request. The request's command name is given as the argument to describe. For example,

ESC X describe apropos CR

describes the "apropos" extended request.

ESC X make-wall-chart
Puts into a buffer a listing of all the currently defined requests, and what keys invoke them in the current buffer. This buffer can be dprinted for a convenient wall chart of Emacs requests.

Error Recovery

^G command-quit
Quits out of the current minibuffer prompt, if any, and rings the bell (or beeps). Can be used to exit a minibuffer you did not intend to get into, or just to tell when Emacs has "caught up."

^X^G ignore-prefix
Flushes a prefix character. Used when a prefix character such as ^X is entered by accident; causes an audible signal to indicate that the ^X^G has been executed. Unlike ^G, ^X^G does not exit the minibuffer.

^Z^G ignore-prefix
Flushes a prefix character. Used when a prefix character such as ^Z is entered by accident; causes an audible signal to indicate that the ^Z^G has been executed. Unlike ^G, ^Z^G does not exit the minibuffer.

ESC ^G ignore-prefix
Flushes a prefix character. Used when a prefix character such as ESC is entered by accident; causes an audible signal to indicate that the ESC ^G has been executed. Unlike ^G, ESC ^G does not exit the minibuffer.

New Lines/Blank Lines

carriage return (CR) new-line
Inserts a newline character into the buffer at the current point, ending the current line, and starting a new one. If entered in the middle of a line, breaks the line at the current point. If the next line is blank, i.e., was made by a single CR or ^O, CR just goes to it and does not insert a newline, except in the case of the last blank line before a nonblank line. If there is a fill prefix (see ^X.), CR inserts it after any newline character it inserts. A ^M is equivalent to a carriage return.

^O open-space
Opens up space by putting a newline ahead of the current point. Pushes all lines of the buffer below the current line down one. With a positive numeric argument, e.g., ^U^U^O, opens up the specified number of lines (16 in this case). See ^X^O to remove (extra) blank lines.

^X^O delete-blank-lines
Deletes blank lines around cursor: gets rid of vertical white space. If issued on a blank line, leaves one blank line; otherwise, deletes all blank lines after this line's end. See ^O.

ESC ^O split-line
Breaks the line at this point, shearing it vertically. Puts the text to the right of the cursor on the next line, with enough indentation so that it is still in the same place horizontally, i.e., the same column. This can be undone by ^U ESC ^.

Indentation and White Space

ESC \ delete-white-sides
Deletes all white space characters on the current line adjacent to the character at the cursor. A white space character is a space, a tab, a formfeed, or a vertical tab.

ESC M skip-over-indentation
Moves the cursor to the first non-white space (i.e., not tab, space, formfeed, or vertical tab) position on this line. In other words, skips over the indentation on this line.

^X. set-fill-prefix
Sets fill prefix in this buffer to be whatever is between the beginning of the line and the cursor (spaces and characters). The fill prefix is inserted automatically by CR, autofill, and runoff-fill-paragraph (ESC Q) into lines after the first line in the buffer. If the cursor is at the beginning of the line when ^X. is issued, the fill prefix is reset (i.e., there is no fill prefix). It can be used to establish a left margin.

^XF set-fill-column
Sets the fill column in the current buffer to be the horizontal position where the cursor is now. The fill column is the "right margin" used by ESC Q to fill and adjust text, by fill mode to fill and adjust text, and by ESC S to determine where to center. The fill column is the first column in which text is not to be placed. The new value of the fill column is printed out in the minibuffer. If a positive numeric argument is given, e.g., ^U 72 ^XF, the fill column is set to that value.

ESC I tab-to-previous-columns
Indents the current point (and rest of line) to line up with the next column to the right in the previous nonblank line. That column begins at the printed character after the first white space character to the right of the current point. Successive ESC Is line the point up in that manner across a line. With a numeric argument, ESC I "unindents" the point.

ESC CR cret-and-indent-relative
Does a carriage return and ESC I. Thus, if the current line is indented, ESC CR ends it and starts a new line, indented the same as the line just ended (the original line). If the original line is not indented, the new line starts under the first character of the second word of the original line. Use this request while you are typing an indented body of text. ESC ^M is equivalent to ESC CR.

ESC ^I indent-to-fill-prefix
Deletes the indentation (leading white space) of the current line, and replaces it with the fill prefix in this buffer, which can be set by ^X..

ESC ^ (caret, not control key) delete-line-indentation
Deletes all white space at the beginning of this line and then merges it with the previous line. With a positive numeric argument, e.g., ^U ESC^, does a ^N first, effectively connecting the next line to this one, without the next line's indentation.

^X^I indent-rigidly
Indents all lines in the region defined by the point and the mark by the amount specified by the numeric argument. The argument can be negative to unindent. All lines having any characters within the region are indented.

Comments

^X; set-comment-column
Sets the comment column in this buffer to the horizontal position where the cursor is now. With a positive numeric argument, sets the comment column at the specified column.

ESC ; indent-for-comment
Searches for this line's comment. If one exists, indents it to the comment column in this buffer (see ^X;). If one does not exist, starts one at the comment column on this line. Uses the comment prefix to search for an old one or start a new one. See also ESC X set-comment-prefix.

^Z; kill-comment
Removes the comment and the white space preceding it from the current line. The deleted text is saved on the kill ring, accessible to ^Y. The text is saved in such a way that following ^Ks and other forward-killing requests merge properly with the deleted text.

ESC N down-comment-line
Properly indents the comment on the next line, or puts a comment on the next line if one is not there already. Effectively the same as ^N ESC . See ESC .

ESC P prev-comment-line
Properly indents the comment of the previous line, or puts one on the previous line if one is not there already. Effectively the same as ^P ESC . See ESC .

ESC X set-comment-prefix "string"
Sets the comment prefix in this buffer. This is usually set automatically by entering a major mode. The comment prefix is given as an argument to this request, in quotes. The comment prefix is used by ESC ;, ESC N, and ESC P to find comments, and start them.

Formatting

^X. set-fill-prefix
Sets fill prefix in this buffer to be whatever is between the beginning of the line and the cursor (spaces and characters). The fill prefix is inserted automatically by CR, autofill, and runoff-fill-paragraph (ESC Q) into lines after the first line in the buffer. If the cursor is at the beginning of the line when ^X. is issued, the fill prefix is reset (i.e., there is no fill prefix). It can be used to establish a left margin.

^XF set-fill-column
Sets the fill column in the current buffer to be the horizontal position where the cursor is now. The fill column is the "right margin" used by ESC Q to fill and adjust text, by fill mode to fill and adjust text, and by ESC S to determine where to center. The fill column is the first column in which text is not to be placed. The new value of the fill column is printed out in the minibuffer. If a positive numeric argument is given, e.g., ^U 72 ^XF, the fill column is set to that value.

ESC S center-line
Centers the current line, according to the fill column set by ^XF.

empty line, this is the same as the beginning of the line.

^N next-line-command
Positions to the next line of the buffer. If on the last line, appends a new empty line to the bottom of the buffer, and positions to the beginning (and end) of it. Successive ^Ns and ^Ps try to maintain the same horizontal position. Repeats with a positive numeric argument; performs ^Ps and repeats with a negative numeric argument.

^P prev-line-command
Moves to previous line of the buffer. Beeps as for ^G if on first line of the buffer. Attempts to maintain the same horizontal position; successive ^P's and ^N's try to maintain the original horizontal position. Repeats with a positive numeric argument; moves to next line and repeats with a negative numeric argument.

^X= linecounter
Displays in the minibuffer the number of lines in this buffer, the line number (the first line is line 1) of the line the cursor is on, and the dprint column position.

ESC G go-to-line-number
Goes to a given line, specified by line number, from the top of the buffer. The positive numeric argument specifies the line number. For instance, ESC 25 ESC G goes to line 25.

@ kill-to-beginning-of-line
Kills all the text to the left of the cursor on the current line. The killed text is saved on the kill ring, and may be retrieved with ^Y.

^K kill-lines
Kills to end of line; when already at end of line, deletes the linefeed (merges lines). If on an empty line, deletes it. If given a positive numeric argument, deletes that many lines, starting from the current point on the current line. Successive ^Ks merge killed text on the kill ring.

Words

ESC B backward-word
Goes backward one word. If in the middle of a word, goes to before the beginning of that word. Skips backward over all white space to get to the next word. Underscores and backspaces count as parts of words. Repeats with a positive numeric argument;

goes forward and repeats with a negative numeric argument.

ESC F forward-word
Goes forward one word. If in the middle of a word, moves to the end of the current word. Leaves point immediately after that word. Passes over all punctuation and white space before the word. Underscores and backspaces count as parts of words. Repeats with a positive numeric argument; moves backward and repeats with a negative numeric argument.

ESC # rubout-word
Deletes the word to the left of the current point. More specifically, deletes going backward, deleting characters until the beginning of a word. Successive words deleted with ESC # are merged and can be retrieved with one ^Y. Repeats with a positive numeric argument; deletes forward and repeats with a negative numeric argument.

ESC \177 rubout-word
Deletes the word to the left of the current point. More specifically, deletes going backward, deleting characters until the beginning of a word. With a positive numeric argument, deletes the specified number of words. Deletes forward the specified number of words with a negative numeric argument. Successive words deleted with ESC \177 are merged and may be retrieved with one ^Y.

ESC D delete-word
Deletes the word to the right of the current point. More specifically, deletes forward, deleting all white space and punctuation and characters until the end of the next word. Repeats with a positive numeric argument; deletes backward and repeats with a negative numeric argument.

ESC C capitalize-initial-word
Capitalizes the initial letter of a word, e.g., Word. If the cursor is in a word or immediately after a word, capitalizes the first letter of that word. Otherwise, acts on the next word. Leaves cursor immediately after the word capitalized.

ESC L lower-case-word
Converts a word to all lowercase, e.g., word. If the cursor is in a word or immediately after a word, lowercases that word. Otherwise, lowercases the next word. Leaves cursor immediately after the word acted upon.

^XESC escape-dont-exit-minibuf
Is the same as ESC, and can be used for all requests beginning with ESC, and numeric arguments. However, can be used in the minibuffer when typing ESC would terminate the minibuffer, as in some of the special search strings.

ESC X extended-command
Prompts for the name and arguments of an extended request in the minibuffer, terminated by CR. To find out about an extended command, type:

ESC X describe <name-of-command> CR

Macros

^X(begin-macro-collection
Starts learning all that follows as a macro, until ^X) or an error occurs. All requests and input between ^X(and ^X) are remembered as a macro, which can be executed by ^XE, or saved and assigned to a key by ESC X save-macro, and displayed by ^X*.

^X) end-macro-collection
Ends a macro definition. The requests and input typed since ^X(become the "last macro defined" for ^XE. If given a numeric argument, re-executes the defined macro as ^XE does (see that request). See ^X(for what a macro is.

^X* show-last-or-current-macro
Displays the requests (as keystrokes, e.g., ^A, esc-B, etc.) in the last macro defined (see ^X(and ^X)). If given a positive numeric argument, e.g., ^U^X*, displays the keystrokes and command names.

^XE execute-last-editor-macro
Executes the last macro defined (by ^X(and ^X)), one or many times depending on the numeric argument to this request.

With:

No argument
Executes it once.

0(i.e., ESC 0^XE)
Executes it over and over, pausing after each execution. Type a space to go on to the next, CR or ^G to stop repeating.

1-9999
Does it that many times.

9999-infinity
Does it until an error occurs.

^XQ macro-query
Queries the user during the execution of a macro so that he can:

continue execution by typing a space

stop execution by typing a ^G

restart execution from the beginning by typing a CR or other character

This request can only be used by including it in a macro definition. Thus:

^X(^S form CR ^XQ ESC U ^X)

locates occurrences of "form" and queries the user before uppercasing them.

ESC X save-macro
Saves a macro, assigning it to a key. Invoke save-macro after a macro has been defined while still in the same buffer. Prompts for a command name to assign to the macro, and a key. A null response for the key does not assign it to any key; ESC X set-key can be used later. When a key has been assigned, this key invokes that macro; it takes arguments identical to ^XE. Assigned keys are only valid while in the buffer in which they were assigned; command names are valid in any buffer.

ESC X edit-macros
Produces a symbolic list of all keyboard macros defined in the current buffer and places it in a new buffer called emacs-macros. The keyboard macros may then be written out for later loading, edited, redefined, or compiled into Lisp code. See Appendix D for full information.

ESC X show-macro <macro-name>
Displays an editor macro (defined with ^X(and ^X) the same as show-last-editor-macro does, but takes the name assigned to the macro (by ESC X save-macro) as an argument.

Characters (Moving by/Deleting)

^B backward-char
Moves backward one character in the buffer. Tabs and the newline characters at the ends of lines count as one character. Beeps as for ^G at the beginning of the buffer. Repeats with a positive numeric argument; moves forward and repeats with a negative numeric argument.

^F forward-char
Moves forward one character. Tabs and newlines count as one character each. Beeps as for ^G at the end of the buffer. Repeats with a positive numeric argument; moves backward and repeats with a negative numeric argument.

rubout-char
Deletes the previous character (before the cursor, which is usually the last character typed). Note that # deletes the character to the left of the cursor, while ^D deletes the character at the cursor. Repeats with a positive numeric argument, deletes the character at the cursor and repeats with a negative numeric argument.

\177 (delete key) rubout-char
Deletes the previous character (before the cursor, which is usually the last character typed). Note that \177 deletes the character to the left of the cursor, while ^D deletes the character at the cursor. Repeats with a positive numeric argument; deletes the character at the cursor and repeats with a negative numeric argument.

^D delete-char
Deletes the character to the right of the current point. This is the character on which the cursor sits. Moves the rest of the line one to the left, closing up the space. Deleting a newline at the end of a line merges lines. Repeats with a positive numeric argument; deletes the previous character and repeats with a negative numeric argument.

Lines (Moving in and by/Deleting)

^A go-to-beginning-of-line
Positions to the beginning of the current line of the buffer, i.e., right before the first character.

^E go-to-end-of-line
Positions to the end of the current line, i.e., after the last character and before the linefeed. On an empty line, this is the same as the beginning of the line.

^N next-line-command
Positions to the next line of the buffer. If on the last line, appends a new empty line to the bottom of the buffer, and positions to the beginning (and end) of it. Successive ^Ns and ^Ps try to maintain the same horizontal position. Repeats with a positive numeric argument; performs ^Ps and repeats with a negative numeric argument.

^P prev-line-command
Moves to previous line of the buffer. Beeps as for ^G if on first line of the buffer. Attempts to maintain the same horizontal position; successive ^P's and ^N's try to maintain the original horizontal position. Repeats with a positive numeric argument; moves to next line and repeats with a negative numeric argument.

^X= linecounter
Displays in the minibuffer the number of lines in this buffer, the line number (the first line is line 1) of the line the cursor is on, and the dprint column position.

ESC G go-to-line-number
Goes to a given line, specified by line number, from the top of the buffer. The positive numeric argument specifies the line number. For instance, ESC 25 ESC G goes to line 25.

@ kill-to-beginning-of-line
Kills all the text to the left of the cursor on the current line. The killed text is saved on the kill ring, and may be retrieved with ^Y.

^K kill-lines
Kills to end of line; when already at end of line, deletes the linefeed (merges lines). If on an empty line, deletes it. If given a positive numeric argument, deletes that many lines, starting from the current point on the current line. Successive ^Ks merge killed text on the kill ring.

Words

ESC B backward-word
Goes backward one word. If in the middle of a word, goes to before the beginning of that word. Skips backward over all white space to get to the next word. Underscores and backspaces count as parts of words. Repeats with a positive numeric argument; goes forward and repeats with a negative numeric argument.

ESC F forward-word
Goes forward one word. If in the middle of a word, moves to the end of the current word. Leaves point immediately after that word. Passes over all punctuation and white space before the word. Underscores and backspaces count as parts of words. Repeats with a positive numeric argument; moves backward and repeats with a negative numeric argument.

ESC # rubout-word
Deletes the word to the left of the current point. More specifically, deletes going backward, deleting characters until the beginning of a word. Successive words deleted with ESC # are merged and can be retrieved with one ^Y. Repeats with a positive numeric argument; deletes forward and repeats with a negative numeric argument.

ESC \177 rubout-word
Deletes the word to the left of the current point. More specifically, deletes going backward, deleting characters until the beginning of a word. With a positive numeric argument, deletes the specified number of words. Deletes forward the specified number of words with a negative numeric argument. Successive words deleted with ESC \177 are merged and may be retrieved with one ^Y.

ESC D delete-word
Deletes the word to the right of the current point. More specifically, deletes forward, deleting all white space and punctuation and characters until the end of the next word. Repeats with a positive numeric argument; deletes backward and repeats with a negative numeric argument.

ESC C capitalize-initial-word
Capitalizes the initial letter of a word, e.g., Word. If the cursor is in a word or immediately after a word, capitalizes the first letter of that word. Otherwise, acts on the next word. Leaves cursor immediately after the word capitalized. Acts on the next N words with a positive numeric argument; acts on the previous N words with a negative numeric argument.

ESC L lower-case-word
Converts a word to all lowercase, e.g., word. If the cursor is in a word or immediately after a word, lowercases that word. Otherwise, lowercases the next word. Leaves cursor immediately after the word acted upon. Lowercases the the next N words with a positive numeric argument; acts on the previous N words with a negative numeric argument.

ESC U upper-case-word
Converts a word to all uppercase, e.g., WORD. If the cursor is in a word or immediately after a word, uppercases that word. Otherwise, uppercases the next word. Leaves cursor immediately after the word acted upon. Uppercases the next N words with a positive numeric argument; acts on the previous N words with a negative numeric argument.

ESC T twiddle-words
Transposes (interchanges) the last two words typed, e.g., "like I ESC T Multics because...," becomes "I like Multics because..." If you are currently in the middle of a word, the cursor goes to the end of the word before the transformation.

ESC _ underline-word
Canonically underlines a word. If the cursor is in a word or immediately after it, underlines that word. Otherwise, underlines the next word. Leaves the cursor immediately after the underlined word. Although the underlined word looks peculiar on the screen, it is correct.

^Z_ remove-underlining-from-word
Removes underlining from the current word; the rules for selecting which word are the same as those used by uppercase-word (see ESC U).

Sentences

ESC A backward-sentence
Goes to the beginning of the current sentence, i.e., just before the first letter. If already at the beginning of a sentence, goes to the beginning of the previous sentence. The beginning of the first word after a blank line always counts as the beginning of a sentence. Repeats with a positive numeric argument; goes forward and repeats with a negative numeric argument.

ESC E forward-sentence
Moves forward to the end of this sentence. If at the end of a sentence, moves forward to the end of the next sentence. Ends of paragraphs are implicitly ends of sentences, whether or not an end-of-sentence punctuation (period, question mark, exclamation point) appears. Repeats with a positive numeric argument; moves backward and repeats with a negative numeric argument.

^X# kill-backward-sentence
Kills backward to the beginning of this sentence; kills as much of the sentence as thus far typed. Successive kills via ^X# and other reverse-killing requests (e.g., ESC #) merge, and may be retrieved with one ^Y. Repeats with a positive numeric argument; kills forward and repeats with a negative numeric argument.

^X\177 kill-backward-sentence
Kills backward to the beginning of this sentence; kills as much of the sentence as thus far typed. Successive kills via ^X\177 and other reverse-killing commands (e.g., ESC #) merge, and may be retrieved with one ^Y. Repeats with a positive numeric argument; kills forward and repeats with a negative numeric argument.

ESC K kill-to-end-of-sentence
Deletes text going forward from the cursor to the end of the current sentence. If at the end of a sentence, deletes forward to the end of the next sentence. Sentences and other text killed consecutively in this fashion are merged, and may be retrieved with a single ^Y. With a positive numeric argument, kills the specified number of sentences. Kills backward the specified number of sentences with a negative numeric argument.

Paragraphs

ESC [beginning-of-paragraph
Moves to the beginning of the current paragraph. If already at the beginning of a paragraph, moves to the beginning of the previous paragraph. The beginning of a paragraph is the beginning of the first line of the paragraph. The definition of paragraphs is controlled by the paragraph-definition-type option: if 1, blank lines separate paragraphs; if 2, an indented line starts a paragraph. The Multics runoff or compose command's control lines count as individual paragraphs. Repeats with a positive numeric argument; moves forward and repeats with a negative numeric argument.

ESC] end-of-paragraph
Moves to the end of the current paragraph. If at the end of a paragraph, moves to the end of the next paragraph. The end of a paragraph is the end of the last line of the paragraph. The definition of paragraphs is controlled by the paragraph-definition-type option: if 1, blank lines separate paragraphs; if 2, an indented line starts a paragraph. The Multics runoff or compose command's control lines count as paragraphs. Repeats with a positive numeric argument; moves backward and repeats with a negative numeric argument.

ESC H mark-paragraph
Puts the mark at the beginning of the current paragraph; puts the cursor at the end of the current paragraph. See ESC [for a definition of paragraphs.

Screens

^V next-screen
Displays next screenful of the current buffer. Leaves cursor at upper left hand corner of screen. With a positive numeric argument, pages forward the specified number of screensful and displays it; with a negative numeric argument, moves backward the specified number of screensful (previous screens) and displays it.

ESC V prev-screen
Displays the previous screen (one back) of this buffer, leaving cursor at the top of it. With a positive numeric argument, moves backward the specified number of screensful and displays it; with a negative numeric argument, moves forward the specified number of screensful and displays it.

^L redisplay-command
Clears the screen, and displays the current window of the current buffer, centered about the current line. Useful if your screen is messed up by messages (preventable with ESC X accept-msgs), non-Emacs output, etc. With a positive numeric argument, moves current line to that many lines below top of screen; ESC 0^L or ESC 1^L moves current line to top, for example. With a negative numeric argument, moves the current line to that many lines above the bottom of the screen; ESC -1^L moves to the bottom of the screen, ESC -2^L moves to two lines from the bottom, etc.

ESC R move-to-screen-edge
Moves to top, bottom, or other point on screen. ESC 1 ESC R or ESC 0 ESC R moves to the top line of the screen, ESC 6 ESC R moves to 6 lines from the top, ^U^U^U ESC R or ESC -1 ESC R moves to the bottom. ESC -2 ESC R moves to the second line from the bottom, etc. Leaves the cursor at the start of the selected line.

^Z^V scroll-current-window
Scrolls the current window up a line, with the cursor maintaining its position relative to the text. With a positive numeric argument, e.g., ^U^Z^V, scrolls up the specified number of lines. With a negative numeric argument, e.g. ESC -3 ^Z^V, scrolls down the specified number of lines.

Buffers

ESC < go-to-beginning-of-buffer
Moves to the beginning of the current buffer, i.e., before the first character in the buffer at the top of the document being edited.

ESC > go-to-end-of-buffer
Moves to the end of the current buffer, i.e., after the last character or newline character (if the buffer ends with a newline) at the bottom of the document being edited.

^XB select-buffer
Switches to another buffer. Prompts for the name of that buffer, terminated with CR. If that buffer does not already exist, it is created. All key bindings, fill column, comment column, comment prefix, etc., associated with that buffer are put in effect. The last point that you were at in that buffer becomes the current point. Responding to ^XB's prompt with only a CR goes to the last buffer you were in.

^X^B list-buffers
Produces listing of buffers and their pathnames. A ">" marks buffer you came from, "*" says buffer is modified since it was last read or written. Proceed with editing, or type linefeed to refresh screen.

^XH mark-whole-buffer
Puts the mark at the end of the buffer and the cursor at the beginning. This "marks" the whole buffer, so that ^W deletes it, etc. The linefeed at the end of the buffer is not in the marked region, but ^XH ^W ^XB ... ^Y effectively moves a whole buffer.

^XK kill-buffer
Kills (destroys) a buffer. Prompts for the buffer's name, terminated by CR. A response of CR kills the current buffer. Buffers can be killed to conserve storage, to prevent their appearance in buffer listings, or to prevent being queried when quitting with ^X^C.

ESC ~ unmodify-buffer
Marks the current buffer as not modified. Emacs does not mention this buffer when querying before quitting the editor. This is useful after accidentally modifying a buffer which you only intended to examine.

^Z^B edit-buffers
Enters the buffer editor. If ^Z^B is given no argument, the buffer editor sets up its display in the current window. If given a positive numeric argument, e.g., ^U ^Z^B, the buffer editor finds some other appropriate window (if in two-or-more-window mode) to set itself up in. See Section 16 for full information on the buffer editor.

Multiple Windows

^XØ remove-window
Removes the window in which the cursor appears from the screen. The cursor is moved to the window that had been visited just before the current window was entered (that is, the next-most-recently visited window). The space occupied by the departing window is divided among its neighbors. With a positive numeric argument, removes the window specified, where the topmost window is 1.

^X1 expand-window-to-whole-screen
Expands the window in which the cursor appears to fill the whole screen; all other windows are removed. This in essence returns to "one window mode" from having any number of windows. The cursor retains its position in the text.

^X2 create-new-window-and-go-there
Creates a new window at the bottom of the screen, redividing the screen equally among all the windows. The cursor moves to the new window, which has a default buffer name created from its window number.

^X3 create-new-window-and-stay-here
Creates a new window at the bottom of the screen, redividing the screen equally among all the windows. The cursor remains where it is. The new window, which has a default buffer name created from its window number, becomes the "least-recently visited window."

^X4 select-another-window
Moves the cursor to the least-recently visited window on the screen. That window then becomes the most-recently visited. Thus, successive applications of ^X4 visit all windows on the screen. This is a good request to use when you want to visit some new buffer or file, but not overwrite windows containing information you have been looking at recently. With a positive numeric argument, e.g., ESC 3 ^X4, goes to that window, i.e., window 3.

^XO select-other-window
Moves the cursor to the other window (in two-window mode), or to the previous window, implicitly (usually) switching buffers. The mode line is updated to reflect the new buffer. The cursor appears at the last point it was in the new window. In general, the cursor enters the window used last, immediately before the current window was entered, so this request switches you back and forth between the two most-recently used windows.

ESC ^V page-other-window
Valid only when more than one window exists. Displays the next screenful of the other window's buffer (i.e., the one the cursor is not now in). With a positive numeric argument, pages forward the specified number of screensful, and displays it. With a negative numeric argument, pages the other window backward. When more than two windows are in use, the next most recently visited window is considered to be the "other window". Very useful for "paging" through compiler diagnostics while editing a program.

^Z^W edit-windows
Enters the window editor to create, realign, destroy, or visit windows. If ^Z^W is given no argument, the window editor sets up its display in the current window. If given a numeric argument, e.g., ^U ^Z^W, the window editor finds some appropriate window to set itself up in. See Section 16 for full information on the window editor.

Mail/Messages

^XM send-mail
Enters the Emacs mailer (RMAIL) to compose outgoing mail. This request prompts for the mail subject terminated by CR. For full information on reading and sending mail, see Appendix B.

^XR rmail
Enters the Emacs RMAIL subsystem to read mail. Without a numeric argument, uses your default mailbox. With a positive numeric argument, e.g., ^U ^XR, prompts for a mailbox name, which can take a form like Washington.States, Salter, >udd>Sales>Complaints, etc. The first message in the mailbox is placed in a buffer in RMAIL mode. Type "q" to exit RMAIL and delete all mail queued for deletion during RMAIL. Refer to Appendix B for full information on RMAIL and reading and sending mail.

ESC X accept-messages
Accepts Multics interactive terminal messages into Emacs buffers, one buffer per correspondent. Messages are displayed as a local display as they arrive. All correspondence to and from each correspondent is maintained in a buffer named "Messages from <PersonName>." In such a buffer, carriage return is bound to respond-from-buffer, which transmits messages when you type a line into that buffer. In these buffers, conversations "transcript" as with the Multics send_message command in input mode (see the MPM Commands). The following key bindings are set up globally by accept-messages (see Appendix F for more information):

^X: message-response-command
Responds to last sender from minibuffer.

^X' go-to-new-message-buffer
Goes to a message buffer.

^X` send-a-message
sends a message, like ^X:, but to anyone you specify.

^X~ repeat-last-message
displays the last message again.

ESC X accept-messages-path
The ESC X accept-messages-path request allows receipt of messages in mailboxes different from your default mailbox. The request requires an argument, which is either a mailbox pathname, a Person_name (for sites using the ARPANet mail daemon), or a Person.Project. Up to 50 mailboxes can be accepting messages in a process.

Typing Shortcuts

^C re-execute-command
Reexecutes the last keystroke (request), other than ^J or ^C. Useful for skipping successive words, etc. Repeats with a positive numeric argument. Reexecutes search requests using the same search string. Reexecutes extended requests, using the same arguments (if any).

^T twiddle-chars
Transposes (interchanges) the last two characters typed, e.g., I like Mutl^Tics becua^Tse..., becomes "I like Multics because..."

^U multiplier
When not followed by a positive number, multiplies the next request by 4 for each use, e.g., ^U^D deletes 4 characters. Typing ^U^U^D deletes 16. With a positive number, uses the number, e.g., ^U13x inserts an x 13 times. A ^U is considered a positive numeric argument; however, ^U -6, for example, is an argument of -6.

^\ undo-prefix
Used as a prefix to another Emacs request to reverse the usual action of the request it precedes. For example, ^X_ underlines an entire region; ^\^X_ de-underlines the region.

ESC T twiddle-words
Transposes (interchanges) the last two words typed, e.g., "like I ESC T Multics because...," becomes "I like Multics because..." If you are currently in the middle of a word, the cursor goes to the end of the word before the transformation.

ESC SPACE complete-command
Used by the ^XB, ^XK, and ESC X requests to complete a minibuffer response. You can type enough of a minibuffer response to unambiguously indicate a known buffer name (for ^XB and ^XK) or extended request name (ESC X), then type ESC SPACE to complete the minibuffer response automatically. If ESC SPACE cannot complete the response because more than one completion is possible, Emacs types an error message. When the ESC X opt cmp:allow-ambiguous option is on (which it is by default), ESC SPACE attempts to complete your minibuffer response even if several possible completions exist, i.e., even if your typed input does not unambiguously specify the completion. In this case, ESC SPACE completes the response with the first match in the current list of completions. Typing another ESC SPACE after a successful completion removes the one just inserted and inserts the next match found; ^U ESC SPACE displays the current completions. If no more possible completions exist, the terminal bell rings and the last completion displayed is removed.

ESC X setab <abbrev1> <expansion1> <abbrevn> <expansionn>
Defines one word as an abbreviation for another, for Speedtype mode. For instance:

ESC X setab edr editor

defines edr as the abbreviation for editor. Accepts multiple pairs of arguments. If the second string (the thing being abbreviated) is many words, or has special characters in it, put it in quotes. The abbreviation can be 4 characters or less.

The pairs entered during one or more of these commands are cumulative for the entire Emacs session. The only entry that overrides a previous entry is one that redefines an earlier abbreviation. To turn an abbreviation off, redefine it to itself, i.e., to turn off the abbreviation above, type:

ESC X setab edr edr

ESC X speedtype
Enters Speedtype minor mode in this buffer. Speedtype allows words to be used as abbreviations for other words. ESC X setab defines abbreviations. When a space or punctuation mark (period, comma, colon, or semicolon) is typed after an abbreviation, the abbreviation is removed from the text and replaced by its expansion. Precede punctuation or spaces with ^Q to deliberately avoid Speedtype expansion when in this mode.

ESC X speedtypeoff
Turns off speedtype mode in this buffer, if it is on.

Programming Modes

ESC X alm-mode
Enters ALM major mode in this buffer. ALM mode consists of many requests and variable settings suitable for the creation and editing of ALM programs (see Appendix C).

ESC X electric-alm-mode
Enters Electric ALM mode in the current buffer. Electric ALM mode is a variant of ALM mode in which colons and carriage returns are automatic.

ESC X electric-pl1-mode
Enters Electric PL/I mode in the current buffer. Electric PL/I mode is a variant of PL/I mode in which semicolons and colons have violent automatic "electric" action which may be disturbing to some, but useful to others. See Appendix C.

ESC X fortran-mode
Enters FORTRAN major mode in this buffer. FORTRAN mode consists of many requests and variable settings suitable for the creation and editing of FORTRAN programs. See Appendix C for a list of the requests and a description of this mode. You can issue the request ESC X apropos fortran CR in a FORTRAN mode buffer for a list of relevant requests in this mode.

ESC X fundamental-mode
Enters Fundamental major mode, the mode (set of key bindings and variable settings) that all buffers start out in. This can be used to "undo" any other major mode that you may have set.

ESC X lisp-mode
Enters Lisp major mode in this buffer. Lisp mode consists of many requests and variable settings suitable for the creation and editing of Lisp programs. See Appendix C for a list of the requests and a description of this mode.

ESC X pl1-mode
Enters PL/I major mode in this buffer. PL/I mode consists of many requests and variable settings suitable for the creation and editing of PL/I programs. See Appendix C for a list of the requests and a description of this mode. You can issue the request ESC X apropos pl1 CR in a PL/I mode buffer for a list of relevant requests in this mode.

ESC X set-compile-options "option string"
In language modes that support ESC ^C for compile-buffer, e.g., PL/I, FORTRAN, sets non-default compilation options to be given to the appropriate compiler.

ESC X set-compiler <compiler>
Sets the name of the compiler to be used by the compile-buffer request (usually ESC ^C) in those language modes that compile buffers this way, e.g., PL/I, FORTRAN. The single argument to ESC X set-compiler is the compiler name.

ESC X ldebug
Enters a "Lisp Top Level" buffer in Lisp Debug mode. Forms typed into this buffer are evaluated and the value is displayed by placing it in this buffer. When ESC X ldebug has been used, all Lisp errors in Emacs trap into this buffer. ESC P restarts a break. See the Extension Writers' Guide for more information.

ESC X text-mode
This major mode can be used for editing text files. It automatically puts you into fill mode, sets the comment column at 0, the comment prefix to "", and changes the ESC ^ request's action so that ESC ^ inserts a space between the two lines that it has forced together. If find-file-set-modes is on, .runoff or .compin-suffixed files instate text mode when read into Emacs.

^XV view-lines
For printing terminals, prints the current line. With a positive non-zero numeric argument, prints the specified number of lines, beginning with the current and continuing on down. With a negative numeric argument, prints the specified number of preceding lines. Leaves you after them, unless argument is 1 or not supplied, in which case it leaves you on the current line, after printing it. ESC 0^XV views the region (between cursor and the mark). Thus, ^XH ESC 0^XV prints the whole buffer.

^Z^L redisplay-this-line
Prints only the current line.

^X^T toggle-redisplay
Turns off all screen updating until the next ^X^T, ^G (not ^X^G or ^Z^G), or error happens. This request can be used on slow terminals with no insert/delete facilities to avoid excessive printing time for operations such as typing in the middle of a line.

^O open-space
Opens up space by putting a newline ahead of the current point. Pushes all lines of the buffer below the current line down one. With a positive numeric argument, e.g., ^U^U^O, opens up the specified number of lines (16 in this case). See ^X^O to remove (extra) blank lines.

Extension Writing

ESC ESC eval-lisp-line
Prompts for a string for Lisp to evaluate; puts a pair of parentheses around it, evaluates it in Lisp (with ibase = 8), and prints out the Lisp value in the minibuffer (base = 8, *nopoint nil). To get a variable value, use ESC ESC progn <varname> CR. If the ESC X opt eval:eval option is off, ESC ESC is bound to the extended-command request.

ESC X ldebug
Enters a "Lisp Top Level" buffer in Lisp Debug mode. Forms typed into this buffer are evaluated and the value is displayed by placing it in this buffer. When ESC X ldebug has been used, all Lisp errors in Emacs trap into this buffer. ESC P restarts a break. See the Extension Writers' Guide for more information.

ESC X loadfile <path>
Loads a private Emacs extension package into the editor. The argument is its pathname. See the Extension Writers' Guide for more information.

ESC X loadlib <library>
Loads an extension package into Emacs. Normally, such packages are "autoloaded" when requests in them are invoked, but from time to time, new, experimental, or highly specialized packages may require being loaded in this way. The single argument is the name of the package to be loaded. Loading a package makes the requests in it available. See the Extension Writers' Guide for more information.

Additional Optional Settings

ESC X opt
Sets internal flags and defaults, each of which have names. Takes three forms:

opt list
Lists all options and settings.

opt NAME VALUE
Sets option value.

opt status NAME
Reports setting of one option.

Where:
NAME is an option name and VALUE is an acceptable value for the named option. Values may be on, off, or numbers, depending on the option. Code, such as start-ups, can set these Lisp variables (on/off => t/nil). Current options are:

add-newline
When on, adds a newline to the end of the buffer being written out, if is does not already end with one. Default is on.

autoload-inform
Prints a minibuffer message, "Autoloading <function name>...", whenever Emacs performs an automatic load, and notifies you when autoload is completed. Default is off.

check-newline
When on, queries user writing a buffer out (with ^X^W or ^X^S) if the buffer does not end in a newline. Default is off.

command-bell
When set to a real number, performs a user notification (rings the bell) whenever a command takes longer than that many real seconds to complete. Default is off.

command-bell-count
: When set to an integer, rings the bell that many times during user notification (see command-bell above). If set to a symbol, that symbol is called as a function (hook) with the number of seconds used by the last command as an argument. Default is off.

cmp:allow-ambiguous
: When on, allows ambiguous completions for the complete-command request, ESC SPACE. Successive ESC SPACE invocations find new completions in the current completion list. If no more possible completions exist, the bell sounds and the last completion displayed is removed. Default is on.

default-comment-column
: Sets comment column for new buffers (0 origin). Default is 60.

This page intentionally left blank.

default-fill-column
Sets fill column for new buffers. Default is 78.

display-ctlchar-with-^
Causes control characters to print as ^P instead of \020. Default is off.

eval:assume-atom
When on, considers a string given to the eval-lisp-line request without spaces or parentheses as an atom instead of a function. Default is off.

eval:correct-errors
When on, corrects syntax errors (including unbalanced parentheses, vertical bars, and double quotes) in eval-lisp-line input lines. When off, does not attempt syntax correction. Default is off.

eval:eval
When on, ESC ESC is bound to the eval-lisp-line request. When off, ESC ESC is bound to the extended-command request. Default is off.

eval:prinlength
Limits the length of list structures printed by the eval-lisp-line request. When off, any length is allowed. Default is 6.

eval:prinlevel
Limits the depth of list structures printed by the eval-lisp-line request. When off, any depth is allowed. Default is 3.

fill-messages
When on, formats (fills) messages as ESC Q does. Default is on.

find-file-check-dtcm
When on, ^X^F causes date-time-contents-modified of the file to be compared to the time buffer was last read or written. If the buffer contains an old version of the file, Emacs prompts for a specified aaction. Default is on.

find-file-set-modes
When on, ^X^F reads the file into an appropriate major mode buffer (according to the last component), e.g., ^X^F foo.pl1 sets PL/I mode. Default is on.

gratuitous-marks
When on, a successful ESC >, ESC <, ^S, or ^R request sets a mark at the current position. Default is off.

kill-ring-max-size
Sets the kill ring size. Maximum is 20; default is 10.

meter-commands
If set on, prints in the minibuffer the number of real seconds used after completion of each command. If set to a symbol, acts like command-bell-count above, but independent of command-bell. Default is off.

no-minibuffer-<>
Suppresses the display of the angle brackets at the termination of your minibuffer input. Default is off.

paragraph-definition-type
1 = only blank lines separate paragraphs.
2 (default) = blank lines or leading white space separate paragraphs.

pop-up-windows
When on, ^XB, ^X^F, ^X^E, etc., find an appropriate place on the screen to put up a window as opposed to replacing contents of current window (experimental; see Appendix H). Default is off.

quit-on-break
When on, pressing the BREAK (or ATTN) key causes Emacs to abort, instead of just pushing a Multics level. Default is off.

read-file-force
When on, ^X^R reads the requested file into a modified buffer, without pause for verification. Default is off.

remember-empty-response
This option is used by the ESC ^Y request; when off, last-minibuffer-response is not set to blank when a blank response is given. Default is off.

rdis-whitespace-optimize
Avoids printing white space when clever terminal control would go faster. Default is on.

rdis-wosclr-opt
Wipes out screen lines before filling screen. Try it both ways to see what this means. Default is off.

rmail-header-format
Controls the header of the mail being displayed. It can have one of the following values (which are the standard mail system formatting modes):

brief-formatting-mode
Specifies that the brief form of the header be included in the message.

default-formatting-mode
Specifies that the default form of the header be included in the message. This value gets rid of redundant or unnecessary fields. This is the default value.

long-formatting-mode
Specifies that the long form of the header be included in the message.

none-formatting-mode
Specifies that the header be excluded from the message.

NOTES: Since Lisp accepts dashes and PL/I accepts underscores, either is allowed as a delimiter (e.g., "brief_formatting_mode" is the same as "brief-formatting-mode").

You can use the Lisp "setq" command to change these options in your start_up.emacs before RMAIL has been loaded. For example:

```
(setq rmail-header-format
  'default-formatting-mode)
```

rmail-reply-include-authors
Includes the authors of the original message, or the addresses in the original message's "Reply-To" field, as primary recipients of the reply. Default is on.

rmail-reply-include-recipients
Includes the primary, secondary, and blind recipients of the original message as secondary and blind recipients of the reply. Default is off.

rmail-reply-include-self
Includes the user as a recipient of the reply message if he is the author or recipient of the original message. Default is off.

rmail-request-acknowledgment
Requests an acknowledgment when sending mail. Default is off.

rmail-send-acknowledgment
Sends an acknowledgment after reading a piece of mail for which an acknowledgment has been requested by the sender. Default is on.

rmail-original-yank-indent
Sets the indentation of retrieved original mail. Default is 4.

rubout-tabs-into-spaces
When on, turns a tab into one minus the number of spaces necessary to reach the current column when deleting characters backward (with # or the delete key). In effect, it makes rubout-char always delete one visible character position. Default is off. (This option has an effect only when rubout-char is called from the keyboard, not from inside Lisp code.)

save-same-file-check-dtcm
When on, ^X^S and ^X^W with no pathname cause date-time-contents-modified of the file to be compared to the time buffer was last read or written. If the file has been modified since the buffer was last read or written, Emacs queries whether to proceed with the save request. Default is on.

screen-overlap
Sets the number of lines from the current screenful of text that should appear in the next screenful displayed by a ^V or ESC V request. Default is 1.

short-message-accept
When on, messages are not put on the screen in local display; a message of the form, "Messages received from Green.ACRE" appears under the mode line instead. Default is off.

suppress-backspace-display
Suppresses the display of backspaces. Causes underlined "foo" to print as "_f_o_o". Default is off.

suppress-ctlchar-display
Suppresses the display of control characters. Any character which would print as \NNN (except \177) is not displayed. Default is off.

suppress-minibuffer
Suppresses all Emacs output to the minibuffer, e.g., error messages such as "search fails," and "no default pathname for this buffer". Prompts are not suppressed. Default is off.

suppress-remarks
When on, suppresses all remarks to the minibuffer, such as notifications of writing and reading files. Default is off.

suppress-rubout-display
Suppresses the display of rubout characters. Causes \177 to never be displayed. Useful when reading ALM listing segments. Default is off.

track-eol-opt
When on, a ^N or ^P at end of line sticks to ends of lines. Default is off.

underline-whitespace
When on, underlines white space within the region being underlined by the underline-region request (^X_). Default is off.

write-file-overwrite
When on, ^X^W with pathname overwrites a file of the same name, without pause for verification. Default is off.

In addition, ESC X opt sets several options for use in PL/I major mode. They are listed below and described in the "Programming Language Modes" section:

pl1-comment-column
pl1-comment-column-delta
pl1-comment-style
pl1-compile-options
pl1-dcl-column
pl1-dcl-style
pl1-first-column
pl1-indentation
pl1-inding-style
pl1-line-length

ESC X option
Is the same as ESC X opt.

ESC X set-minibuffer-size <size>
Sets the size of the minibuffer/prompting area on the screen to any value. The single argument to ESC X set-minibuffer-size is the decimal number of lines that should be devoted to this function, from 1 to 6. The default is two. With many-line minibuffers, many messages and errors may appear at once. Use ESC X reset-minibuffer-size to reset the minibuffer size to its default of two lines.

ESC X reset-minibuffer-size <size>
Resets the size of the minibuffer/prompting area to its default of two lines. See ESC X set-minibuffer-size.

ESC X set-screen-size <size>
Sets the size of the main editing area (the area above the mode line). The default is all of the area above the mode line. The decimal argument to ESC X set-screen-size is the number of lines in the main editing area. ESC X reset-screen-size resets the main editing area size to its default value. ESC X set-screen-size is usually used to reduce the amount of redisplay at low terminal speeds.

ESC X reset-screen-size <size>
Resets the size of the main editing area of the screen to its default, namely, all of the space above the mode line. See ESC X set-screen-size.

ESC X set-key <keyname> <command-name>
Assigns key bindings in the current buffer. Takes two arguments, the key name and the command name. Makes that key execute that request in this buffer. The command name is what describe, apropos, or make-wall-chart give; the key name can be anything like the names in this documentation, e.g., ^X, ^x, ESC ESC, ^Xq, control-p, c-p, meta-f, ESC ^f, CR, ^X ^F, ^X CR, \177, #, A, ^P, etc. See Section 15 for a full description of acceptable key names.

ESC X set-permanent-key <keyname> <command-name>
Sets permanent (default in all buffers) key bindings. Otherwise, works exactly like ESC X set-key.

ESC X set-search-mode <search-mode>
Sets the bindings of ^S and ^R to invoke several different forms of searching. The argument is the search mode, and can be:

string
Searches forward/backward (depending on whether you are searching with ^S/^R) for the exact character string typed (the default).

regular-expression or regexp
Interprets the search string as a regular expression (^S now behaves as ESC /). Reverse searching is also available.

character
Searches for the single character typed, (end with an ESC) unless that character is one of the following, which are interpreted specially:

^A
invokes the ITS string search request, or reverse ITS string search request (^R). (See ITS-string-search for more details.)

^G
aborts the search

^J
finds next/beginning of a line, or previous/end of line (^R).

^M

finds next/beginning of a line, or previous/end of line (^R).

^Q

reads another character and searches for it regardless of its value. You should use ^Q when searching for control characters.

^R

reverses the direction of the search and reads another character (or searches again for the default string if ^R is the character read). The character read is then processed as if it were read by ^R, or ^S when searching with ^R.

^S

searches for the current default string. The default is displayed in the minibuffer. If the search succeeds, the cursor is left after/before the character(s) found.

^_

displays a description of the allowable responses (i.e., prints this list).

ITS-string

A ^S or ^R reads characters and either adds them to the search string, or performs some action as specified by the character. All non-control characters are added to the search string. With the exception of the control characters listed below, all the control characters are invalid and are ignored. The special control characters recognized by ^S and/or ^R are:

\177

removes the last character from the search string. If there are no characters in the search string, the search aborts.

#

is the same as \177.

^B

changes the starting point for searches requested by ^S or ESC. When ^S is entered, searches are made starting from the current cursor position. A ^B starts searches from the beginning of the buffer. A subsequent ^B starts searches from the current cursor position, etc.

^D

replaces the search string with the current default search string, and "rotates" the list of defaults. The default search string is the last string used by any other search request, or by the use of ^S (see below) to this request. A ^D permits you to "walk" through the last sixteen strings you searched for to find the one you wish to search for again.

^G

aborts the search.

^L

redisplays the screen.

^Q

reads a character and adds it to the search string regardless of its nature. A ^Q is the only way to place control characters into the search string.

^R

reverses the direction of the search. Any characters read after the ^R are processed by the ^R request, or by the ^S request if you are using ^R.

^S

searches forward/backward for the current search string. If found, the cursor is placed after/before the string. In any event, the current search string is pushed onto the top of the list of default search strings.

^Y

adds the current default search string to the search string typed so far. Thus, ^S^Y^Y^S searches forward for two successive occurrences of the default search string and ^R^Y^Y^R searches backward.

ESC

searches forward/backward for the current search string and then exits the ^S/^R request. If the previous character typed to ^S was ^S, ESC only exits and does not search. The search done by ESC is identical to that done by a ^S.

^_

displays a description of the allowable responses (i.e., prints this list).

incremental

Searches by character forward/backward (^S or ^R) for a string character as you type the string. The string accumulates in the minibuffer. That is to say, the cursor moves in the main window finding more and more accurate matches for the string you are typing as you specify further characters of it. The general idea is to find the string you are looking for before you finish typing it, at which point you type ESC. The cursor is left after/before the string found in the buffer. It is recommended only for high speed lines: try it in order to get the general idea. The following characters have special meaning when typing the search string to incremental-search; all normal "printing" characters are searched for:

^S

for ^S, finds the next match for the current search string. If the current search string is empty, retrieves the default search string set by the last search command.

^S

for ^R, reverses the direction of search, entering incremental-search. The first occurrence of the search string is found going forward in the buffer.

ESC

ends the incremental search to allow other requests to be typed. It is very important to remember to type ESC.

^G

aborts the incremental search, and returns to the place from which it was started.

or \177
removes the last character from the search string, and moves the cursor to the place it was before you typed that character.

^L
redisplays

^Q
quotes the next character, i.e., puts it into the search string literally.

^R
for ^S reverses direction of searching. This enters reverse-incremental-search, searching backward for the first occurrence of the current search string (which is always at the current point in the buffer).

^R
for ^R, repeats the search, going backward, for the current search string, i.e., finds the next occurrence of it. If the current search string is empty, retrieves the default search string.

^_
displays a description of the allowable responses (i.e., prints this list).

If an attempt is made to add a character to the search string which produces a search string that cannot be found, Emacs rings the terminal bell and removes the character from the minibuffer, leaving the cursor in place. If this occurs during a macro execution, however, a normal Emacs ^G is done, aborting macro execution.

All searches prompt, telling the type of search that has been invoked.

APPENDIX A

THE MULTICS EMACS COMMAND

SYNTAX AS A COMMAND:

emacs {-control_args} {paths}

FUNCTION: Enters the Emacs text editor, which has a large repertoire of requests for editing and formatting text and programs. Emacs is a display-oriented editor designed for use on CRT terminals. Several modes of operation for special applications (e.g., RMAIL, PL/I, FORTRAN) are provided; the default mode entered is Fundamental major mode, whose requests are listed below.

ARGUMENTS:

paths
are pathname(s) of segments to be read in. Each is put into its own appropriately named buffer (a find-file operation is done on each path).

CONTROL ARGUMENTS:

-terminal_type STR, -ttp STR
specifies your terminal type to Emacs, where STR is any recognized editor terminal type or the pathname of a control segment to be loaded. The terminal type is set permanently; changing the Multics terminal type during a login session does not affect the type "remembered" by Emacs. If STR is not a recognized type, Emacs queries you after entry, providing a list of recognized types.

-reset
specifies that Emacs disregard the terminal type set by the -ttp control argument and set it in accord with the Multics terminal type instead.

-query
causes Emacs to query the user for a terminal type without checking the Multics terminal type first. The query response can be any recognized editor terminal type.

-line_length N, -ll N
overrides the terminal controller default for line length, and sets it to the number of characters specified by N.

-page_length N, -pl N
overrides the terminal controller default for page length, and sets it to the number of lines specified by N.

-line_speed N
indicates linespeed to obtain proper padding (for ARPANet users), where N is the output line baud rate in bits/second.

-no_startup, -ns
prevents use of the user's startup (start_up.emacs).

-macros path, -mc path
loads the segment, specified by path, as Lisp, so that features therein are available.

-apply function_name arg1 arg2...argi, -ap function_name arg1 arg2...argi
evaluates (function_name 'arg1 'arg2...'argi), where the args are arguments to the named Lisp function (e.g., an Emacs request). This is valuable for constructing abbrevs. This control argument must be the last argument.

NOTES: None of the terminal_type control arguments (-ttp, -reset, -query, -line_speed, -page_length, -line_length) is generally necessary; they are only used for solving various communications problems.

Argument evaluation order, in relation to pathnames, macro pathnames, and -apply arguments, is:

1. Evaluate pathnames and macro pathnames in the order given in the command line.

2. Evaluate the -apply arguments.

ALPHABETIZED LIST OF FUNDAMENTAL MODE REQUESTS

The following is a list of Emacs Fundamental mode requests, alphabetized by the last character. Everything preceding the last character of each request is arranged in this suborder, under that last character: ^, ESC, ESC^, ^X, ^X^, ^Z, ^Z^. Extended requests are listed separately at the end.

#	rubout-char
ESC #	rubout-word
^X#	kill-backward-sentence
@	kill-to-beginning-of-line
^@	set-or-pop-the-mark
^Z^@	set-named-mark
CR	new-line
ESC CR	cret-and-indent-relative
^XCR	eval-multics-command-line
ESC	escape
ESC ESC	eval-lisp-line
^XESC	escape-dont-exit-minibuf
\	escape-char
^\	undo-prefix
ESC \	delete-white-sides
\177	rubout-char
ESC \177	rubout-word
^X\177	kill-backward-sentence
^X(	begin-macro-collection
^X)	end-macro-collection
^X*	show-last-or-current-macro
^X.	set-fill-prefix
ESC ;	indent-for-comment
^X;	set-comment-column
^Z;	kill-comment
^X=	linecounter
ESC %	query-replace
^_	help-on-tap
ESC _	underline-word
^X_	underline-region
^Z_	remove-underlining-from-word

ESC /	regexp-search-command
ESC <	go-to-beginning-of-buffer
ESC >	go-to-end-of-buffer
ESC ?	describe-key
ESC [	beginning-of-paragraph
ESC]	end-of-paragraph
ESC ^	delete-line-indentation
ESC ~	unmodify-buffer
^XØ	remove-window
^X1	expand-window-to-whole-screen
^X2	create-new-window-and-go-there
^X3	create-new-window-and-stay-there
^X4	select-another-window
^A	go-to-beginning-of-line
ESC A	backward-sentence
^B	backward-char
ESC B	backword-word
ESC ^B	balance-parens-backward
^XB	select-buffer
^X^B	list-buffers
^Z^B	edit-buffers
^C	re-execute-command
ESC C	capitalize-initial-word
^X^C	quit-the-editor (quit)
^D	delete-char
ESC D	delete-word
^XD	edit-dir
^E	go-to-end-of-line
ESC E	forward-sentence
^XE	execute-last-editor-macro
^X^E	comout-command
^F	forward-char
ESC F	forward-word
ESC ^F	balance-parens-forward
^XF	set-fill-column
^X^F	find-file
^ZF	object-mode-find-file
^Z^F	get-filename

^G	command-quit
ESC G	go-to-line-number
ESC ^G	ignore-prefix
^XG	get-variable
^X^G	ignore prefix
^ZG	go-to-named-mark
^Z^G	ignore-prefix
ESC H	mark-paragraph
^XH	mark-whole-buffer
ESC I	tab-to-previous-columns
ESC ^I	indent-to-fill-prefix
^XI	insert-file
^X^I	indent-rigidly
^J	noop
^K	kill-lines
ESC K	kill-to-end-of-sentence
^XK	kill-buffer
^L	redisplay-command
ESC L	lower-case-word
^X^L	lower-case-region
^Z^L	redisplay-this-line
ESC M	skip-over-indentation
^XM	send-mail
^N	next-line-command
ESC N	down-comment-line
^O	open-space
ESC ^O	split-line
^XO	select-another-window
^X^O	delete-blank-lines
^P	prev-line-command
ESC P	prev-comment-line

^Q	quote-char
ESC Q	runoff-fill-paragraph
^XQ	macro-query
^R	reverse-string-search
ESC R	move-to-screen-edge
^XR	rmail
^X^R	read-file
^S	string-search
ESC S	center-line
^XS	global-print-command
^X^S	save-same-file
^T	twiddle-chars
ESC T	twiddle-words
^X^T	toggle-redisplay
^U	multiplier
ESC U	upper-case-word
^X^U	upper-case-region
^V	next-screen
ESC V	prev-screen
ESC ^V	page-other-window
^XV	view-lines
^Z^V	scroll-current-window
^W	wipe-region
ESC W	copy-region
ESC ^W	merge-last-kills-with-next
^XW	multi-word-search
^X^W	write-file
^Z^W	edit-windows
ESC X	extended-command
^XX	put-variable
^X^X	exchange-point-and-mark
^Y	yank
ESC Y	wipe-this-and-yank-previous
ESC ^Y	yank-minibuf
^Z^Z	signalquit

Extended Requests

```
ESC X       accept-messages
ESC X       accept-messages-path <address>
ESC X       alm-mode
ESC X       apropos <string>
ESC X       describe <extended-request>
ESC X       edit-macros
ESC X       electric-alm-mode
ESC X       electric-pl1-mode
ESC X       filloff
ESC X       fillon
ESC X       fortran-mode
ESC X       fundamental-mode
ESC X       ldebug
ESC X       lisp-mode
ESC X       list-named-marks
ESC X       loadfile <path>
ESC X       loadlib <library>
ESC X       lvars
ESC X       make-wall-chart
ESC X       opt <option>
ESC X       option <option>
ESC X       overwrite-mode
ESC X       overwrite-mode-off
ESC X       pl1-mode
ESC X       replace
ESC X       reset-minibuffer-size <size>
ESC X       reset-screen-size <size>
ESC X       runoff-fill-region
ESC X       save-macro
ESC X       set-comment-prefix  "string"
ESC X       set-compile-options "option string"
ESC X       set-compiler <compiler>
ESC X       set-key <keyname> <command-name>
ESC X       set-minibuffer-size <size>
ESC X       set-permanent-key <keyname> <command-name>
ESC X       set-screen-size <size>
ESC X       set-search-mode <search-mode>
ESC X       setab <abbrev1> <expansion1> <abbrevn> <expansion>
ESC X       show-macro <macro-name>
ESC X       speedtype
ESC X       speedtypeoff
```

APPENDIX B

EMACS MAIL

The Emacs mail system provides a facility for reading, sending, and responding to Multics mail within Emacs, utilizing the standard Emacs features and the interfaces of the Multics mail system. There are two basic functions, sending mail and reading mail.

SENDING MAIL

^XM

The Emacs request for sending mail is:

^XM, send-mail

Issuing this request puts you in MAIL mode and prompts for a "Subject," which should be supplied and terminated by a carriage return. This subject is incorporated into the buffer name, so it should be short. A buffer is formatted with the mail in it, header prefabricated. The buffer is placed in an available window, as with ^X^E (comout-command). Fill mode is turned on with a fill column of 72. The buffer is now in MAIL mode, which defines the following requests:

^XA mail-append
Goes to the end of the body of the mail. Use this to enter the text after you have set the destination, or to go back to the text after editing some header field.

^XT mail-to
Goes to the end of the "To:" line, to add a recipient. You are left here when the MAIL buffer is entered, to enter the first recipient. Then use ^XA to continue. Separate recipients (like all header fields) with commas, e.g.,

To: Smith.Sales, Consultant.c

^XF mail-from
Goes to the end of the "From:" line, to edit it or add more sender's names.

^XJ mail-subject
Goes to the end of the "Subject:" line, to edit it.

^XC mail-cc
Goes to the end of the "Cc:" (carbon copy recipients) line, making one if there is none, so that you can type in the destination of a carbon copy recipient.

^XY mail-reply-to
Generates a "Reply-To" field, if none exists, and goes to it. The destination put here is used for replies if a recipient of your mail uses RMAIL mode (or another mail system) to reply automatically to your message.

^X^A request-acknowledgment
Requests an acknowledgment when sending mail.

^X^S send-the-mail
Sends the buffer to the recipients specified in the header. A message appears as a local display to confirm that the mail is sent (2 linefeeds to restore display).

ESC ^F forward-mail-field
Moves forward one field (recipient, cc recipient, etc.) on this (header) line. Circles around at end.

ESC ^B backward-mail-field
Moves backward one field (recipient, cc recipient, etc.) on this (header) line. Circles around at end.

ESC ^D delete-mail-field
Deletes, including necessary commas, the single header item (recipient, etc.) that the cursor is on.

^XL rmail-logger-append
Logs the message into a file, placing it at the end, separated by a formfeed. Prompts for the pathname of the log file with a numeric argument, or the first time it is used. Otherwise, uses the same file last used by ^XL or ^XP.

^XP rmail-logger-prepend
Same as ^XL, but puts message at the front of the file.

The Emacs send-mail request recognizes the following forms of mail addresses. In the syntax of messages, braces ({}) must be coded as specified; brackets ([]) are used to indicate optional information and must not be entered as part of the request.

1. Person_id
   Multics Person_id only. This form is possible only if the Person_id is an entry in the mail table.

2. Person_id at SYSTEM [address-route]
   or
   Person_id@SYSTEM [address-route]
   Identifies an address on another computer system. "Person_id" is the user(s) to receive the message and is not interpreted by the local system, and "SYSTEM" is the name of the foreign system where the address is located. If [address-route] is given, the foreign system name does not have to be known to the local system; if not supplied, SYSTEM is the primary name of the foreign system as specified in the local system's network information table (NIT). This format address is valid only for systems connected to the ARPA network.

3. Person_id.Project_id
   Specifies either a user's default mailbox (>udd>Project_id>Person_id.mbx) or a user's logbox (>udd>Project_id>Person_id>Person_id.sv.mbx).

4. {forum PATH}
   Identifies a Forum meeting by pathname, where "forum" is a fixed field and "PATH" is a variable field that specifies the absolute pathname of the meeting, excluding the "control" suffix.

5. {list PATH}
   Identifies a mailing list, where "list" is a fixed field and "PATH" is a variable field that specifies the absolute pathname of the mailing list segment, excluding the "mls" suffix.

6. {logbox}
   Is specified only in the syntax of new messages and identifies the user's logbox (>udd>Project_id>Person_id>Person_id.sv.mbx). When the message is delivered, the syntax of this address is converted to the Person_id.Project_id format described above.

7. {mbx PATH}
   Identifies an arbitrary mailbox by pathname. "PATH" is the absolute pathname of the mailbox excluding the "mbx" suffix.

8. {save PATH}
Appears only in the syntax of new messages and identifies a savebox. "PATH" is the absolute pathname of the savebox excluding the "sv.mbx" suffix.

Parenthetical comments in destinations are ignored, thus:

Muhammad (I am the Greatest) Ali at (the) WBA

gets sent to Muhammad Ali at site WBA. Quote processing is done, and a field between angle brackets (<>) makes all outside it, in a given address, a comment.

Net mail sending is done via the Network Mailer Daemon; net connect access is NOT required. The Mailer Daemon does not send an acknowledgment message. Your name is given as:

From: Smith

or, if this site is on the ARPANET:

From: Smith at MULTIX

If RMAIL knows your real name, your name is given as:

From: Sarah M. Smith <Smith at MULTIX>

The mail system looks in the default value segment for Person_id.full_name._ or full_name._ to find your real name; otherwise, the "From:" field is set to your mail table entry.

READING MAIL

^XR

The Emacs request for reading mail is:

^XR, rmail

By default, mail is read from your personal default mailbox. With a positive numeric argument (.e.g, ^U), ^XR prompts for the "mailbox name." This may take any of the forms:

```
Person_id.Project_id
<pathname>              (with or without ".mbx" suffix)
Person_id               (if Person_id is an entry in
                         the system mail table)

{logbox}
{save PATH}
{mbx PATH}
```

If you have no mail in the selected mailbox, a message is issued to this effect. Otherwise, the first message in the mailbox is displayed in a buffer, in RMAIL mode. This buffer is read-only. Now, when you are perusing the mailbox and then use ^XB or ^Z^B to go to another buffer, a subsequent ^XR or ^U^XR returns to the mailbox currently active without giving a warning or notification. To change mailboxes, you must issue a q request while in RMAIL mode to end reading of one mailbox. With the next ^XR or ^U^XR, you access another mailbox. The following extra requests (all standard requests work here, too) apply in RMAIL mode(these are mostly not control characters, but regular characters). Numeric arguments for these requests do not require that ESC precede them, e.g., 3g goes to the third message.

n rmail-go-forward
Moves on to the next message. Accepts a numeric argument.

p rmail-go-backward
Moves back to the previous message. Accepts a numeric argument.

l rmail-go-last-msg
Moves to the last message in your mailbox.

g rmail-go-command
Moves to the message number specified by the numeric argument, e.g., 3g to go to message #3, or 1g to go to the first message.

j rmail-go-command
Same as g.

d rmail-queue-delete-forward
Deletes (i.e., queues for deletion when rmail is exited) this message, moves on to next undeleted message.

D rmail-queue-delete-backward
Same as d, but moves backward.

u rmail-undelete
Brings back the last (stacked) deleted message.

c rmail-copy
Copies the message to some other mailbox. Prompts for a save mailbox name. It takes a pathname.

q rmail-quit
Quits out of rmail, returning to buffer from which rmail was invoked, deleting all messages marked for deletion.

s rmail-summarize
Summarizes (in a local display) all undeleted messages; may take a little time for full mailboxes.

^XL rmail-logger-append
Logs the message into an ASCII file, placing it at the end of the file. See the description above under the mail-sending requests.

^XP rmail-logger-prepend
Same as ^XL, but "prepends" to the front. See the description above under mail-sending requests.

r rmail-reply
Formats a MAIL mode buffer to reply to the current message, copying the subject (if any), or making one up, and setting up as the destination the sender's "reply-to" address. Responses can be sent to other recipients as well if a numeric argument (e.g., 1r) is supplied. This request is extremely effective in two-window mode, in which case your response is put in one window, the letter you are replying to in the other, and ESC ^V (page-other-window) can be used to "page" the letter as you respond. ESC ^Y can be used while preparing the reply to "yank" the original piece of mail.

m send-mail-from-rmail
Sends mail that is not necessarily a reply (see r). Identical to ^XM, send-mail, but also allows use of ^X^Q, ^X^S, and ESC ^Y (described below).

^X^Q rmail-quit
Quits out of rmail, returning to the buffer from which rmail was invoked, deleting all messages marked for deletion.

Once you have invoked the r request, you can use the MAIL mode requests as well as the standard Emacs requests. In addition, the following three requests are available:

^X^Q return-to-rmail
Returns to RMAIL and its window without sending the message.

^X^S send-from-rmail
Sends the reply and return to RMAIL and its window.

ESC ^Y rmail-yank-mail
Yanks and indents the text and header of the original mail being responded to. Default indentation is 4; can be reset by ESC X opt rmail-original-yank-indent.

It is important to quit (q) out of RMAIL before leaving Emacs; Messages do not actually get deleted unless you quit out of RMAIL (or, equivalently, answer "yes" to "All messages deleted. Quit RMAIL?").

If the rmail-mode-hook Lisp variable is bound by the user, the atomic symbol to which it is bound is called as a function with no arguments before the first message is displayed. This can be used to set rmail-mode key binding.

APPENDIX C

PROGRAMMING LANGUAGE MODES

Emacs, in addition to the Fundamental major mode for general editing tasks, provides four programming language major modes. Modes are sets of key bindings and variable settings; different modes allow more elastic environments for different editing problems. The Lisp, FORTRAN, PL/I, and ALM modes facilitate both programming in, and editing programs written in those languages.

Each of these major modes is entered via an appropriate extended request (see the applicable paragraphs below) or via the ^X^F request. Also, the following Fundamental mode requests that deal with comments have certain language-specific applications.

Fundamental Mode Requests for Programming Use

Most of the Fundamental mode requests listed below deal with comments. Programming languages generally each have a preferred column in which the programmer's comments begin, and this is called the comment column. Since the comments must be distinguishable from the program, they are delimited in some way, either by being begun and ended by certain character strings, or by being restricted to certain columns. The comment prefix is the comment's beginning delimiter. Several of the requests below use the comment prefix to recognize a comment.

These comment requests are also useful in Fundamental mode for preparing two-column text.

^X;

The ^X; request, set-comment-column, sets the comment column in the current buffer at the horizontal position where the cursor is currently located. With a positive numeric argument, sets the comment column at the column number specified. See ESC ; below for additional information.

^Z;

The ^Z; request, kill-comment, removes the comment and white space preceding it from the current line. The deleted text is saved on the kill ring, accessible to Y. The text is saved in such a way that following ^Ks and other forward-killing requests merge properly with the deleted text.

ESC ;

The ESC ; request, indent-for-comment, searches for the current line's comment. If one exists, it indents it to the comment column in this buffer (set by ^X;). If none exists, this request starts one at the comment column on this line. It uses the comment prefix to search for an old one or start a new one (see the extended request, ESC X set-comment-prefix).

ESC N

The ESC N request, down-comment-line, properly indents the comment on the next line, or puts a comment on the next line if none is there already. This is effectively the same as the sequence ^N ESC ; (see ESC ;).

ESC P

The ESC P request, prev-comment-line, properly indents the comment on the previous line, or puts one there if none is there already. This is effectively the same as the sequence ^P ESC ; (see ESC ;).

ESC ^B

The ESC ^B request, balance-parens-backward, skips backward over one set of balanced parentheses. It searches backward until a set of parentheses is found. However, it does not handle quoting or any programming language conventions. This cannot be used in Lisp mode, which has its own ESC ^B function.

ESC ^F

The ESC ^F request, balance-parens-forward, skips forward over one set of balanced parentheses. It searches forward until a set of parentheses is found. However, it does not handle quoting, or any other programming language conventions. This cannot be used in Lisp mode, which has its own more powerful ESC ^F function.

ESC X set-comment-prefix

The set-comment-prefix extended request sets the comment prefix in this buffer. The comment prefix is usually set automatically by entering a major mode. However, it can be set by giving the comment prefix you want as an argument to this request. The argument must be in quotes, and should follow the request. The ESC ;, ESC N, and ESC P requests all use the comment prefix to find and start comments.

ESC X set-compile-options

The set-compile-options extended request sets non-default compilation options to be given to the appropriate compiler in language modes that support ESC ¨C for "compile buffer" (e.g., PL/I, FORTRAN).

ESC X set-compiler

The set-compiler extended request sets the name of the compiler to be used by the compile buffer request (usually ESC ^C) in those language modes that compile buffers this way (e.g., PL/I, FORTRAN). The single argument to ESC X set-compiler is the compiler name. It must follow the request and be quoted.

ESC ESC

The ESC ESC request, eval-lisp-line, prompts for a string for Lisp to evaluate, parenthesizes it, evaluates it in Lisp (with ibase = 8), and displays the Lisp value in the minibuffer (base = 8, *nopoint nil). To get a variable value, type ESC ESC progn <variable_name> CR.

ESC X ldebug

The ldebug extended request enters a "Lisp Top Level" buffer in Lisp Debug mode. Forms typed into this buffer are evaluated and the value is displayed by placing it in this buffer. When ESC X ldebug has been used, all Lisp errors in Emacs trap into this buffer. See the Extension Writers Guide for more information.

ESC X fundamental-mode

The fundamental-mode extended request enters the mode that all buffers start out in. This request allows you to exit any other major mode that you may have entered.

The following paragraphs describe the programming language major modes and their special key bindings and variable settings.

LISP MODE

Lisp mode facilitates the construction and editing of Lisp programs in Multics Emacs. Requests for positioning over Lisp expressions, and indenting and commenting Lisp code are available.

A facility within Emacs for debugging programs is available. It is called LDEBUG, and is described in the Extension Writers' Guide.

Lisp major mode is entered by issuing the ESC X lisp-mode extended request, or by responding to ^X^F (find-file) with any file with a last component name of ".lisp" when the find-file-set-modes option is selected. When in Lisp major mode, the comment column is set to 50 (column 51), and the comment prefix to ";". The Fundamental mode comment requests, ESC P, ESC N, and ESC ; act according to these settings.

The following is the current request repertoire of Lisp mode:

TAB (^I) indent-to-lisp
On a blank or empty line, creates enough leading white space so that the first S-expression typed on this line lines up properly according to conventional Lisp indenting rules. Normally, this means line it up with the start of the previous S-expression, but in other circumstances other actions may be taken. On a non-blank line, readjusts the line's indentation to effect conventional Lisp indenting.

ESC Q lisp-indent-function
Puts point and mark around the current Lisp function (see ESC ^A). For all lines other than the first, re-indents them according to conventional Lisp indentation.

ESC ^A begin-defun
Moves point to the beginning of the current "function". The beginning of a function is defined as right before the last open parenthesis at the left margin.

ESC ^B backward-sexp
Moves backward over exactly one balanced S-expression. All comments, quoted strings, and slashified characters are considered properly. Aborts (and beeps) if unbalanced. Avoid invoking this from inside comments

or quoted strings. Skips trailing open parentheses. Accepts numeric arguments for repetition count.

ESC ^C compile-function
Compiles and loads the current Lisp function via the Multics Lisp Compiler (lcp). Does this by loading lcp into the Emacs environment (the first time it is used in an Emacs invocation), utilizing it, and loadfiling the result. ESC ^C automatically incorporates/compiles the correct version of e-macros.incl.lisp into your environment (the first time) as well. Puts the name of the function compiled on the kill ring, so it can be yanked into an ESC ESC minibuffer for trial. Displays compiler diagnostics as local display. Be careful to write out changes you make and debug via this facility; this is a common trap: you see what you have in front of you "working", and you think you are done. Forms compiled via ESC ^C are treated as though they had been encountered at top level by the compiler; macro definitions, declarations, and side effects from compilation to compilation are all handled correctly. Definitions, macro definitions, and reader macro definitions other than those in e-macros.incl.lisp must be ESC ^Ced explicitly. Information produced during any ESC ^C remains for all future ESC ^Cs in an Emacs invocation; the regnant Emacs environment is used as both the compile and load time environments. Thus, macro and other definitions ESC ^Zed or ESC ESCed are seen by the compiler. See the Exension Writers' Guide for more information.

ESC ^D down-list-level
Goes down one level of list structure. Basically the same as looking forward for an open parenthesis, but it detects and handles Lisp comments, quotes, etc.

ESC ^E end-defun
Goes to right after the last close parentheses of the current function. See begin-defun above for a definition of the current function. Useful to see if function balances parentheses correctly.

ESC ^F forward-sexp
Skips forward over exactly one S-expression, positioning to after the appropriate close parenthesis, or before the appropriate white space. Accepts numeric arguments for repetition count. Skips leading close parentheses. Avoid invoking inside quoted strings or comments.

ESC ^H mark-defun
Puts point and mark around the current function. See begin-defun for a definition of the current function.

ESC ^K kill-sexp
Kills one (or many) S-expressions forward, i.e., from point to the point after that many complete S-expressions. Argument is the number of S-expressions. Merges kills forward.

ESC CR (ESC ^M) lisp-cret-and-indent
Identical to a CR (newline) followed by indent-to-lisp; this is the normal way to terminate an input line in Lisp mode. It puts you on a new line and indents correctly for the next S-expression. Done in the middle of a line, it breaks the line at that point, correctly indenting the S-expression which was to the right of point on the new line.

ESC ^N forward-list
Moves to right after the end of current Lisp list. Basically, the same as searching for a close parenthesis, but detects and handles Lisp comments, quoting, etc.

ESC ^P backward-list
Moves to right before the beginning of the current Lisp list. Basically the same as searching backward for an open parenthesis, but detects and handles Lisp comments, quoting, etc.

ESC ^Q lisp-indent-region
Re-indents all lines (other than the first) in the point-to-mark region for conventional Lisp indentation.

ESC ^R move-defun-to-screen-top
Moves the current function (see begin-defun above for definition) to the top of the current screen, leaving point at function beginning.

ESC ^T mark-sexp
Puts point and mark around the current S-expression. If point is currently before the close parenthesis of a list, sets point and mark around that list. If point is before white space, marks the next S-expression.

ESC V view-defun
Prints out current function: puts point and mark around the current Lisp function (see begin-defun above for a definition), and displays it (prints it out, on printing terminals) as a local display.

ESC ^Z eval-top-level-form
Evaluates the current top level form and displays its value in the minibuffer. A top level form has the same definition as a function. (See begin-defun above for the definition). Loads the file "e-macros.incl.lisp" to ensure the presence of the Emacs macros (see the Extension Writers Guide). This facility is intended for use in debugging extensions, as is compile-function, but runs your code interpreted rather than compiled to aid in debugging. Be careful to write out changes you make and debug via this facility.

ESC (lisp-one-less-paren
Removes one close parenthesis from the end of the last S-expression prior to the current line, and reindents the current line accordingly. Accepts numeric arguments for repetition count. Use this when the line seems indented (automatically) too few levels, probably due to an extra close parenthesis on the previous line.

ESC) lisp-one-more-paren
Adds one close parenthesis to the end of the last S-expression prior to the current line, and reindents the current line accordingly. Accepts numeric arguments for repetition count. Use this when the line seems indented (automatically) too many levels, probably due to a missing close parenthesis on the previous line.

ESC &

Sets an LDEBUG breakpoint at the cursor (see the Extension Writers' Guide for more information).

The following extended request is available in Lisp mode:

ESC X eval-buffer
Evaluates the contents of the buffer and displays the value of the last form in the buffer via the minibuffer. Loads the file "e-macros.incl.lisp" to ensure the presence of the Emacs macros. This request is used to "load" a buffer of Lisp code into the Emacs environment for debugging. The eval-top-level-form function (ESC ^Z) can then be used to "reload" any functions whose definition you change while debugging. Be careful to write out changes you make and debug via these facilities.

FORTRAN Mode

Emacs FORTRAN mode aids in the construction and debugging of FORTRAN programs. Requests are provided for producing comment and continuation cards and for other commonly used formatting operations.

FORTRAN major mode is entered by issuing the ESC X fortran-mode extended request. This mode can also be entered by responding to ^X^F (find-file) with any file with a last component name of ".fortran" when the find-file-set-modes option is selected.

The current list of special requests in Fortran mode is:

Carriage return (CR) new-line
Returns the cursor to column 7. This also inserts comment prefixes ("c ") in the appropriate places. If desired, the cursor can be moved back by hand to delete spaces or comment prefixes.

^I fortran-indent-statement
Causes a tab done in column 1 to tab to column 7. Subsequent tabs move to the usual places. The sequence for successive tabs is 7,10,20,30...

ESC CR (ESC ^M) fortran-continue
Ends the current line when the next line is a continuation. A newline is done, unless the current line is blank or empty, and the prefix " & " is inserted, leaving the cursor in column 9. This is the continuation for standard FORTRAN rather than for Multics.

ESC ; fortran-comment-line
Begins a single comment line. A newline is done, unless the current line is blank or empty , and the prefix "c " inserted, leaving the cursor in column 7. This can be used to end a line when the next line is a comment.

^XC fortran-begin-comment-block
Begins a block of comments. Ends the current line, if any, and inserts a comment block header line. Any future lines added are prefixed by the standard comment prefix "c ". This minor mode is exited by a second ^XC. Notice that ^X^C exits the editor. Don't miss

ESC : fortran-label
Positions a fortran label. Since a line usually starts in column 7, this request is provided to correctly position statement numbers. Type the statement number, then ESC : to place the label in column 1.

ESC ^C compile-buffer
Compiles the buffer. Writes current buffer, if changed, out to its default pathname (as for ^X^S), and then compiles it. Compiler diagnostics are displayed, in the other window if in two-window mode. The extended requests ESC X set-compiler and ESC X set-compile-options can be used to select the compiler and options to be used.

^X^D locate-next-error
Finds the next error; used following a compilation in two-window mode. It scans the compiler output and the source buffer in parallel, pointing a simulated cursor to consecutive errors and placing the real cursor on the line referred to in the error message. This mode terminates itself when you advance past the last error, do another compilation, or exit it by keying ^XT. Since this mode locks the buffer used for compilations, you are strongly advised to exit it when you are finished.

fortran-abbrev-expander
Expands abbreviations. The two characters immediately preceding the cursor when this character is struck are taken as an abbreviation and expanded. The ' may itself be inserted by quoting it with ^Q. These abbreviations are initially supplied:

in	integer	su	subroutine
di	dimension	co	continue
fu	function	re	return
eq	equivalence	ex	external
au	automatic	cn	common
fo	format	im	implicit

Note that, with the exception of cn for common, all of these abbreviations are the first two characters of the word. Other abbreviations may be defined using the extended request ESC X set-fortran-abbrev. The abbreviations co (continue) and re (return) are very special. Since these words almost always have a label and sit on a line by themselves, these abbreviations do an ESC : and a newline, so that typing (in column 7):

```
123co'
```

expands to:

```
123 continue
```

leaving you on the next line. The abbrev fo, for format, does labels, but not newlines. Try them.

In addition, several extended requests are provided to set various parameters. They are:

For comment blocks:

ESC X fortran-set-begin-comment CR
Sets the begin line.

ESC X fortran-set-end-comment CR
Sets the end line.

Both of these requests prompt for the line in the minibuffer. The line supplied is inserted exactly as given, and must therefore include the "c" at the beginning. Default values for these lines are:

```
"c ========================================".
```

These are set independently for buffer.

For compilations:

ESC X set-compiler compiler-name CR
Sets the compiler to be used. Default is "ft".

ESC X set-compile-options options CR
Sets compile options. These are given as on the compiler command. The default is "-tb".

These are set by buffer.

And, for abbreviations:

ESC X set-fortran-abbrev abbrev expansion label eol
can be used to define new abbreviations. In the above, abbrev must be a two character abbreviation that will be replaced with expansion. Arguments must be enclosed in quotes if they contain special characters, including spaces. The optional arguments label and eol cause this abbrev to handle labels and newlines, respectively, just like co and re. The label option does not require a label, but processes it if it is present. Abbreviations are defined globally and apply to all buffers in FORTRAN mode. For example:

```
ESC X set-fortran-abbrev as "common /xyz/ y(100)" eol
```

defines an abbrev, as, that expands to:

```
common /xyz/ y(100)
```

and does a newline, but does not handle labels.

Currently, the recommended debugging method is to do a ^Z^Z and run your program one level up. Return to Emacs with a program_interrupt command.

PL/I MODE

PL/I mode provides an automatic assistance in PL/I progam formatting in the real-time editing context of Multics Emacs. The basic facility provided at this time is that of lining up untyped PL/I statements, although on a one-for-one basis it lines up typed ones, too.

PL/I major mode is entered by issuing the ESC X pl1-mode extended request, or by responding to ^X^F (find-file) with any file with a last component name of ".pl1" when the find-file-set-modes option is selected. It takes a couple of seconds to "load itself." When in PL/I mode, which shows up on the mode line as "PL/I" major mode, the following non-default key bindings apply:

TAB (^I) indent-pl1-statement
Indents this PL/I statement properly (if not yet typed in, tab out to it; otherwise, readjusts its indentation properly).

ESC CR (ESC ^M) pl1-cret-and-indent
Like carriage return and TAB.

ESC ^C compile-buffer
Compiles the buffer. Writes current buffer, if changed, out to its default pathname (as for ^X^S), and then compiles it. Compiler diagnostics are displayed, in the other window if in two-window mode. The extended requests ESC X set-compiler and ESC X set-compile-options can be used to select the compiler and options to be used.

ESC ^D pl1dcl
Tries to find a declaration for the entry point whose name is to the left of the cursor, and inserts it. A library of such entry points exists. If the declaration is not in the library, ESC ^D attempts to figure it out from inbound parameter descriptors in an object segment responding to that name. Can also declare error table entries.

ESC ^H (ESC Backspace) roll-back-pl1-indentation
Deletes 5 columns of indentation. Intended for undenting ends.

ESC TAB (ESC ^I) pl1-tab-one-more-level
Adds 5 columns of indentation. Intended for asserting your own style.

^X^D locate-next-error
Finds next error. This request is used following a compilation in two-window mode. It scans the compiler output and the source buffer in parallel, pointing a simulated cursor to consecutive errors and placing the real cursor on the line referred to in the error message. This mode terminates itself when you advance past the last error, do another compilation, or exit it by keying ^XT. Since this mode locks the buffer used for compilations, you are strongly advised to exit it when you are finished.

ESC SPACE pl1-skip-to-dcl-column
Moves cursor to pl1-dcl-column (set by ESC X opt pl1-dcl-column; see "PL/I Options" below). If the statement already extends beyond this column on the current line, the cursor moves to that column on the next line. If you are already at the declaration column, the cursor goes to the next line for declarations. This request is useful for indenting attributes in declare statements when ESC ^D cannot be used and the pl1-dcl-style option is set to 2.

ESC ^A pl1-backward-statement
Moves backward over PL/I statements. Accepts a numeric argument specifying the number of statements to move backward.

ESC ^E pl1-forward-statement
Moves forward over PL/I statements. Accepts a numeric argument specifying the number of statements to move forward.

ESC * pl1-comment-end
Moves to the end of the line (as determined by ESC X opt pl1-line-length) and places a comment suffix (*/) at the end of a comment line.

^ZD pl1-line-between-procs
Generates a dividing line between major blocks of code in a PL/I program to provide visual separation. The line extends through the pl1-line-length column and is of the form:

```
/* * * * * ... * */
```

If the cursor is at the beginning of a dividing line, ^ZD inserts a new page as well as a new divider. With a numeric argument, ^ZD inserts a divider, a new page, and a second divider.

^XC pl1-comment-box
Starts or ends a comment. When a new comment box is created, fill mode is entered to facilitate typing of comment text (to exit or then reenter fill mode, use ESC X filloff and ESC X fillon). The ESC X opt pl1-line-length option controls the filling of comment lines. A subsequent ^XC exits the comment mode and completes the comment box by placing a comment suffix (*/) at the ends of all box lines. If ^XC is typed to begin a comment while the cursor is already within an existing comment box, new comment lines are inserted above the line on which the cursor is positioned. Filling occurs only for the new lines; old lines remain unchanged.

^ZC pl1-refill-comment-box-region
Fills (fill mode) the comment box lines between, and including, the lines containing the cursor and the mark.

^ZI pl1-include-file-comment-start-end
Generates a comment line at the start and end for PL/I include files. The lines have the form:

```
/* START OF:      xxx.incl.pl1      *  *  *  */
/* END OF:        xxx.incl.pl1      *  *  *  */
```

All the standard comment requests (ESC ;, ESC N, ESC P, etc.) are set for PL/I and observe the comment style in effect (set by ESC X opt pl1-comment-style, described below). ESC N and ESC P accept a numeric argument specifying the number of lines up or down the cursor should move before commenting the line. Word requests (ESC F, ESC N, ESC P, etc.) in PL/I mode buffers consider the dollar sign to be part of a word.

PL/I Options

Several options can be set to give you more flexibility in writing and editing your PL/I programs. They are set by the ESC X opt request; you simply type ESC X opt, and then the option name and value (see the ESC X opt request if you need more information). When in a pl/1-mode buffer, setting a PL/I option changes its value only in the current buffer. When not in a pl1-mode buffer, setting a PL/I option changes the value for all new pl1-mode buffers. Setting these options in a start_up.emacs segment sets the defaults for all your pl1-mode buffers.

The PL/I options are:

```
ESC X opt pl1-comment-column
          pl1-comment-column-delta
          pl1-comment-style
          pl1-compile-options
          pl1-dcl-column
          pl1-dcl-style
          pl1-first-column
          pl1-indentation
          pl1-inding-style
          pl1-line-length
```

The pl1-comment-column option sets the column in which comments start. The default is 61. If non-comment text extends beyond this column when one of the line comment requests is given, then the placement of the comment depends upon the pl1-comment-style.

The pl1-comment-style option controls how comments are handled when non-comment text extends into the pl1-comment-column. It can have the following values:

1 Comment is placed on the current line following the non-comment text.

2 If non-comment text extends beyond pl1-comment-column + pl1-comment-column-delta (another option), then the comment is placed on a new line below the current line. Otherwise, it is placed on the current line.

3 Comment is placed on a new line following the current line.

The default value for pl1-comment-style is 1. The pl1-comment-column-delta option's default value is 10.

The pl1-compile-option option specifies the default compilation options used by ESC ^C to compile the program. The default compiler option is the null string.

The pl1-dcl-style option determines the format of a declaration. It can have the following values:

0 No formatting is performed.

1 Formats like the indent command. It assumes that the word "dcl" begins in column 1, followed by 2 spaces and the name. Lines longer than pl1-line-length are folded, being continued from column 11.

2 Formats like the format_pl1 command with indattr mode. It assumes that dcl is located between columns 1 and 10, and that the name is in column 11. The declaration begins at the column set by the pl1-dcl-column option. Lines linger than pl1-line-length are folded, being continued from pl1-dcl-column + 5.

The default value for pl1-dcl-style is 1. The pl1-dcl-column option's default value is 41.

The pl1-inding-style option provides two styles of indentation, 1 and 2. The general indentation rules followed are:

Any fragment of an incomplete statement gets lined up 5 spaces after the start of that statement. The statement after a DO or BEGIN gets indented 5 times one less than the number of IF's in the DO or BEGIN. In pl1-inding style 2, the statement after an end gets lined up 5 less than the end statement; the first statement in a program gets lined up at column 11 (can be changed by resetting the pl1-first-column option, whose default is 10). Otherwise, each statement lines up with the previous one. The pl1-indentation option sets the indentation increment for successive indentation levels. Its default value is 5.

In style 1, you get:

```
if x = 6 then do;
      bar = 5;
      foo = 6;
end;
```

In style 1, you are expected to line the end up yourself (use ESC ^H) because it is impossible in realtime to predict that an untyped statement is going to be an end. You must undent the end yourself, because the next statement lines up with it. The default value for pl1-inding-style is 1.

In style 2, you get:

```
 if x = 6 then do;
      bar = 5;
      foo = 4;
      end;
next = 17;
```

With style 2, Emacs can figure out the next statement after the end once you have typed it.

The pl1-line-length option controls the length of lines generated by pl1dcl (ESC ^D) and several of the other requests described above. The line length is specified in terms of column positions, with a default value of 112.

There are no known bugs in the mode's PL/I parsing: it can parse any valid PL/I statement, except that multi-dimensional label constants are not supported.

Electric PL/I Mode

A minor mode called "electric PL/I mode" is available, which can be obtained by ESC X electric-mode CR once in PL/I mode, or ESC X electric-pl1-mode CR. To get it by default as your mode for PL/I programs, put the statement:

```
(defprop pl1 electric-pl1-mode suffix-mode)
```

in your start_up.emacs. Some users have found electric PL/I mode overly violent, so it remains an option. It connects semicolon to a function which automatically indents for the next statement after inserting a semicolon; use ^Q; to get a semicolon in without the "electric" action. Also, this action is suppressed if there is a next line, and it is not empty. The "electric semicolon" also moves ends back for you (in inding-style 1), when you type the ";" of the end statement. (Be careful to quote semicolons with ^Q in strings, or you may have problems.) Electric PL/I mode also gives ":" electric action, i.e., indenting after labels.

ALM MODE

ALM mode provides several variable settings suitable for the creation and editing of ALM programs.

ALM major mode is entered by issuing the ESC X alm-mode extended request, or by responding to ^X^F (find-file) with any file with a last component name of ".alm" when the find-file-set-modes option is selected.

In ALM mode:

the comment column is set to 41

the comment prefix is set to null

the fill prefix is set to tab (a carriage return automatically indents to the opcode field of the ALM statement)

In addition, carriage return is treated in such a way that extra fill prefixes and blank lines are deleted whenever possible. ALM mode also removes the indentation preceding labels typed before a colon.

Electric ALM Mode

A minor mode called "electric ALM mode" is available, and can be obtained by ESC X electric-mode CR once in ALM mode, or ESC X electric-alm-mode CR. To get it by default as your mode for ALM programs, put the statement

```
(defprop alm electric-alm-mode suffix-mode)
```

in your start_up.emacs. This mode does automatic colons and carriage returns.

APPENDIX D

MACRO EDIT MODE

The Macro Edit major mode is available for a number of purposes. With it, you can enter a dedicated buffer to:

- display a symbolic file of all named keyboard macros currently defined
- edit the displayed macros
- redefine macros after editing them
- write the macros out to a file, for dprinting, or for using them in later invocations of emacs.

ENTERING MACRO EDIT MODE

You enter the Macro Edit mode by issuing the ESC X edit-macros extended request, or by issuing the ^X^F request to read in a file with the .emacro suffix when the find-file-set-modes option is selected. This puts you in a buffer displaying, in editable form, all the macros that you have saved with ESC X save-macro. The display includes PL/I-like comments (/* comment */). The comment column in this buffer is automatically set to 51, and the ESC;, ESC P, and ESC N comment requests act accordingly. The macro definitions look like this:

```
macro paragraph-stars on ^X9
     esc-] ^XQ ^O "**------------**"
end-macro paragraph-stars
```

The key setting, e.g., "on ^X9," is optional. If present, it sets the key permanently, i.e., in all buffers, to that macro.

EDITING THE MACROS

In Macro Edit mode, the following requests are available for editing the macros (they are designed to parallel Lisp mode):

ESC ^A macedit-find-beginning-of-macdef
Moves to the beginning of the current macro definition.

ESC ^B macedit-backward-term
Moves backward one term in the macro.

ESC ^C macedit-compile-to-lisp
Compiles the macro being pointed at into Lisp (so that you get it permanently incorporated into Emacs). Many cases are not yet handled by the macro compiler, so you should not use this request unless you can verify that the Lisp code is correct.

ESC ^E macedit-find-end-of-macdef
Moves to the end of the current macro definition.

ESC ^F macedit-forward-term
Moves forward one term in the current macro definition.

ESC ^H macedit-mark-whole-macro
Puts point and mark around the current macro definition.

ESC ^K macedit-kill-term
Kills forward to the end of the current (or next) term in the current macro definition.

ESC ^N macedit-forward-macdef
Moves forward to the beginning of the next macro definition.

ESC ^P macedit-backward-macdef
Moves backward to the beginning of the previous macro definition.

ESC ^S macedit-state-keyboard-macro
Prompts for a key and places the definition of the keyboard macro on that key in the buffer at the current point.

REDEFINING MACROS

ESC ^Z and ESC X load-these-macros

After you edit a macro, you can redefine it so that it works according to the new version, rather than the old. You must issue the ESC ^Z request, macedit-take-up-definition, while still in Macro Edit mode. It replaces the old definition of the macro being pointed at with the new definition just edited. If you do not issue this request (or the one below), the old definition continues to apply during this Emacs session.

The ESC X load-these-macros extended request has the same actions and restrictions as ESC ^Z; the difference is that it redefines all the macros in the buffer. So, if you edit more than one macro, this request is more convenient.

WRITING MACROS OUT TO A FILE

If you want to use your macros in later Emacs sessions, or if you want a printed copy of them, you must write the Macro Edit buffer's contents to a file. The ^X^W request does this. If you write the macros out to a file whose suffix is ".emacro," however, subsequent ^X^Fs on that file will automatically read it into Macro Edit mode, saving you a step.

Using Macros Previously Written to a File

ESC X load-macrofile

When you write your macros out to a file, with or without the .emacro suffix, you can reuse the same macros in later Emacs sessions. They can be automatically defined in the current session if you issue the ESC X load-macrofile extended request. This request takes the pathname of the file containing your macro definitions as an argument; type the pathname after typing the command name, and end the prompt with a carriage return.

You can load more than one macrofile and still have the earlier ones effective. Once loaded, these macros can be used in any buffer by using the command name after ESC X (extended-command) or by using the assigned key. They also exist in a buffer whose name is the same as the segment name without the .emacro suffix.

APPENDIX E

USING EMACS ON PRINTING TERMINALS AND GLASS TELETYPES

Emacs was specifically designed for use on intelligent video terminals, but you can use it on printing terminals and "glass teletypes." Glass teletypes have screens, but do not have cursor addressing or the usual display management capabilities e.g., the TELERAY 3700 and Honeywell Model 7700 Visual Information Projection system. If you are accustomed to using Emacs, you may wish to use it on such terminals. Emacs also may have features and extensions that you wish to use that other editors do not offer. If possible, however, you should first learn how to use it on a video terminal.

The usage of Emacs on a printing or glass teletype terminal is designed to be as close to video terminal usage as possible; all Emacs requests and features operate on any type of terminal (with the exception of specifically video-oriented features such as multiple-window mode). Thus, once you learn a sizable number of requests, or have perhaps written some keyboard macros or extensions, you can use them on any type of terminal.

Printing terminals use the print-head or print-wheel (or actual cursor of a glass teletype) as a cursor. The single line of the buffer being edited upon which the point "appears" is always displayed, and the print-head is moved to the position to the right of the point, as is the cursor on a video terminal. Whenever you want to move the cursor to a new line, e.g., with the ^N or ^P requests, Emacs prints that line and repositions the cursor. If you move many lines at once, with a search, ESC G, or ESC <, for example, Emacs displays only the line on which the cursor "stops." If a line longer than the width of the terminal is to be displayed, all portions of it (i.e., the whole buffer line) are displayed. Once a line has been displayed in this manner, the print-head is moved to the point's position. As requests to move the point back and forth are issued, e.g., ^B, ^F, ESC B, ^A, the print-head moves around on the displayed line accordingly. Printing and glass teletype terminals are treated very much like video terminals with a one-line window.

When you invoke emacs on a printing or glass teletype terminal, Emacs prints the mode line. It is reprinted every time it changes. Similarly, the path line is printed every time it changes. Local displays are simply printed out (with a "More?" query when a glass teletype screen has been filled), followed by the reprinting of the current buffer line; no line of dashes and stars appears, and no linefeed is needed. A record of the local display appears on the terminal paper, or scrolls up the screen on the glass teletype. Messages normally destined for the minibuffer are also simply printed out; Emacs prompts are typed on a new line and the responses awaited. Enter the responses as usual, ending with a carriage return as usual. After you supply a response, Emacs displays the current line, and repositions the print-head appropriately.

Typing a ^L, redisplay-command, at any time prints the mode line, path line, and current line.

Since only one buffer line is shown at once, you need some way to view many lines at once. Simply repositioning the cursor does this on video terminals. On printing terminals and glass teletypes, the ^XV request, view-lines, fills this need in a manner similar to the "print" requests of line-oriented editors like edm and qedx. The ^XV request, with no argument, displays, as a local display, the current line. (Try this on a video terminal for fun!) With a numeric argument of zero, e.g., ESC 0 ^XV, it displays a region; thus, ^XH ESC 0 ^XV displays the whole buffer. With any other positive numeric argument, e.g., ^U ^U ^XV, it prints the specified number of lines (16 here) from the current line on down, and leaves you on the next line after them (type another ^XV to see that line printed). With a negative numeric argument, the specified number of lines preceding the current line are printed.

On a video terminal, Emacs keeps the image of the buffer that is on the screen current by erasing and correcting text as requests are issued. Hardcopy terminals obviously cannot erase or correct what is already printed. Instead, whenever a line changes, Emacs performs a linefeed, scrolling the paper vertically, and prints the portion of the line that changed, and all of the line to the right of the change. Thus, deleting a character in the middle of a line prints all of the line following the deleted character on a new line.

On glass teletypes, the modified right-hand portion of the line is simply rewritten in place (this also occurs on video terminals that do not support insert/delete characters), with no linefeed.

Typing or deleting characters in the middle of a line creates a lot of output as the remainder of the line is continually reprinted. The continual repeating, besides being annoying, takes time, even on fully cursor-addressable video terminals that do not support insert/delete characters. Two techniques minimize this problem.

The ^X^T request, toggle-redisplay, suppresses all printing. If you issue a ^X^T, all "updating", including seeing what you type (echoing) is inhibited. Another ^X^T, or a ^G or any error, performs all the updating at once and turns off the suppression of printing. You can see the current line by issuing a ^XV; all editing changes that occurred while printing was suppressed are reflected. However, if you turn screen updating off with the ^X^T request, then ^V, ESC V, and the other window-modifying requests are disabled until you turn screen updating back on (with another ^X^T request or with a ^G request).

The second technique is to issue the ^O request, open-lines, at the point in the line where you wish to make a change. This inserts a newline character, pushing the rest of the line down a line, and leaves you editing at the new "end" of the line. When you finish editing, a ^D or ^K deletes the newline and brings back the last part of the original line, reprinting it also.

Bear in mind that a complex editing operation that affects many lines, e.g., ESC Q or ESC K, places the cursor wherever that particular request leaves it, printing that line out if it was not the last line printed. Other lines may change as well, but they are not printed. As on a video terminal after a request that changes text over a large region of a buffer, the cursor is left as it is positioned at the end of the operation. An important difference is that you do not have a whole windowful of text surrounding the current line with which to reorient yourself.

The best way to discover how Emacs works on printing terminals is to sit down at one and experiment with the various requests. You will soon become accustomed to the editing methods required. If your terminal is not one of the types specifically recognized by Emacs, invoke the emacs command with the "-terminal_type printing" control argument, or type the word "printing" when Emacs asks you for your terminal type.

Notes

Very commonly, use of Emacs printing terminal support is accidental! If you are logged in on a video terminal, your start_up.ec or the person using the terminal before you may have incorrectly or inadvertently specified to Multics that you are using some type of hardcopy terminal. Then, when you invoke emacs, this misinformation is used, and you enter Emacs in printing terminal mode. This is indicated if, upon entry to Emacs, you find that the screen is not cleared, the mode line is displayed and scrolled up, the cursor is left on the line after the mode line, and attempts to clear the screen with ^L repeat these actions. In this case, exit Emacs with ^X^C as usual, and reinvoke emacs with the -query control argument. Emacs will query you for an acceptable terminal type (a ? or null response prints a list of acceptable terminal types), and reset it accordingly for the rest of the login session. Then, edit your start_up.ec if it caused the problem.

A common problem encountered by those using Emacs on a printing terminal for the first time is that of characters appearing twice. This is always the result of a terminal's echoing characters locally. If this happens, exit Emacs and make certain that both your modem and terminal are set for full-duplex operation. For terminals like the TermiNet 300, which have a controllable local printer, Emacs turns the printer off automatically, and this is not necessary. Otherwise, use full duplex (fulldpx) and echoplex modes, with a full duplex connection, modem, and terminal. This should be the case for video terminals as well as for printing and glass teletype terminals.

Every video terminal can operate as a glass teletype. If you have a video terminal for which no support package (CTL) is supplied, glass teletype usage is preferable to printing terminal usage, since lines are corrected by erasing and rewriting. Glass teletype usage is entered by invoking emacs with the -terminal_type glasstty control argument, or by typing the word "glasstty" when Emacs asks you for your terminal type. You can use Emacs this way until a CTL can be constructed for your terminal type (see the Extension Writers' Guide for information on constructing CTLs). In fact, this mode is useful for editing and debugging a new CTL until it works reliably.

APPENDIX F

THE MESSAGE FACILITY

You can receive interactive messages, sent via the Multics send_message commands, while editing in Emacs. The ESC X accept-messages extended request is provided so that those messages, which appear on your screen as a local display, are then conveniently saved in buffers from which you can respond to their senders.

ESC X accept-messages

If you have not issued an ESC X accept-messages request, or included it in your start_up, messages appear on your screen as Multics output, destroying the current display. If you do issue this request, however, it displays each incoming message, causes the terminal to beep, and enters the message into a buffer named "Messages from <Person_name>. All correspondence to and from an individual is maintained in its own separate, appropriately named buffer. This request also provides the following response capabilities.

^X:

To respond to the sender of the last message received, type the ^X: request, message-response. Suppose Sarah Smith just sent you a message. You are prompted for your response to her in the minibuffer:

```
To Smith:
```

Type your message; when you type a carriage return, it is sent to Sarah. With this method, you remain in your current buffer, send the reply from there, automatically enter the reply into the Messages from Smith buffer, and can immediately resume your work.

With a numeric argument, ^X: switches you to the message buffer of the last sender, so you can see the previous messages to and from that person, while you type a reply to them if you wish. Then you can ^XB back to your working buffer.

^X`

The ^X` request, send-a-message, is similar to ^X: but allows you to send a message to anyone. It prompts for a Person_name of someone who already has a message buffer, i.e., they are or have been corresponding with you during the current Emacs session, or a Person.Project (e.g., Smith.Sales) for someone who has not yet been in correspondence with you (or Person at Net-Host-Name, or Person @ Host). It then prompts for a message to send to that person. Again, you have not switched to a message buffer, so can resume work as soon as the message is sent.

^X'

The ^X' request, go-to-new-message-buffer, always switches you to a message buffer. You are prompted for the name of the person whose message buffer you wish to enter:

Messages to/from:

You can type in the Person_name (not Person.Project) of someone who already has a message buffer. You can give a null response to go to the message buffer of the last sender (as with ^X: with a numeric argument). Finally, to switch to the message buffer of a person not currently communicating with you, you give the name in the form of Person.Project, or Person at Net-Host-Name (or Person @ Host). In any of these cases, you then simply start typing your message, and send it with your first carriage return.

When ^X' is given a numeric argument, it lists message buffers (conversations), and the direction of the last interaction. Thus, buffers in which you sent the last message are marked =>, and those in which the other party did are marked <=.

^X~

The ^X~ request, repeat-last-message, repeats the last message as a local display. It is more convenient than ^X' if a message flashes on and off your screen before you were able to read it.

ESC X accept-messages-path

The ESC X accept-messages-path request allows receipt of messages in mailboxes different from your default mailbox. The request requires an argument, which is either a mailbox pathname, a Person_name (for sites using the ARPANet mail daemon), or a Person.Project. Up to 50 mailboxes can be accepting messages in a process.

A further convenience of the message facility is its use with multiple windows. When the message buffers are on display, incoming messages are displayed immediately in their appropriate windows, without appearing as local displays. You can carry on several "conversations" at once, and can write any of the buffers out if you want a record of them.

APPENDIX G

EMACS START-UPS

Emacs can be instructed to execute a sequence of requests at the time it is invoked. This allows you to customize your environment, i.e., to set up things that are not provided by default. You do this with a file called an Emacs start-up. You do not need one: it is optional. If you do not have one, Emacs performs as described in this manual, unless administrators at your site have set up project-wide or system-wide start-ups.

Normally, when you invoke Emacs, it searches first in your home directory for a personal start_up called start_up.emacs. If it doesn't find one, it then looks for >udd>Project_id>start_up.emacs, in case your project administrator has created a start_up for all members of your project. If it doesn't find one there either, it finally looks for >site>start_up.emacs, in case your system administrator has created one for all the users on your system. If you want to do things like enable the Emacs message system every time you use Emacs, you will want to have your own personal start_up, which overrides any global ones. Your best bet in making a start-up is to copy someone else's and modify it. However, instructions for writing start-ups are provided here. (In the sample start_ups throughout this section, multi-line start_ups include a number in the first column of each line. The numbers are not part of the file; they are here simply for reference.)

Here is just about the simplest possible start-up:

```
(accept-messages)
```

If you put this line in a segment called "start_up.emacs" in your home directory, Emacs will accept messages (i.e., activate the Emacs message system described in Appendix F) every time it starts up. The line above invokes the ESC X accept-messages extended request just as though you had typed ESC X accept-messages). The parentheses around this "request name" (i.e., accept-messages) tell Emacs that "this is something to do," i.e., that the name in parentheses is the name of an extended request to be executed.

Here is a slightly more complicated start-up:

```
1  ; Sally's start-up
2
3  (accept-messages)
4  (setq my-personal-name "Sarah M. Smith")
```

Line 1 is a comment. It has no meaning other than to let anyone reading the file know that this is Sally's start-up. The semicolon at the beginning of the line indicates that the rest of the line is to be ignored. Blank lines, such as line 2, also have no meaning, and are ignored. Line 2 is provided simply for readability.

On line 3, Sally activates the Emacs message system. Although this is the most common thing people want to do in start-ups, not everyone wants this to be done, since those just beginning to use Emacs might not know how to use the Emacs message system. They should defer messages while using Emacs, instead of having messages destroy their screens.

On line 4, Sally is telling the Emacs mail system (See Appendix B for a full description of it) what her "personal name" is. This is used by the send-mail request (^XM) for mail headers. When Sally uses RMAIL, she will get header lines like:

```
From:  Sarah M. Smith  <Smith.Sales>
```

in messages she sends. The "setq" is a keyword meaning "set the value of a variable," in this case the variable "my-personal-name." Variables are named boxes in which things are kept (you may have used Emacs text variables, which are manipulated by the ^XX and ^XG requests). The Emacs mail system looks in the box named my-personal-name for your full name, so this is the name of the variable you must supply. Note that Sarah's full name is between quotation marks; you must quote your name, so that its beginning and end can be determined. You must have a statement just like this in your Emacs start-up if you want the Emacs RMAIL system to know your full name when you compose mail. (Alternatively, you can request your site's Emacs expert to place your name in the "rmail-full-name-table," of full names of people at your site.

Once more, this text must appear in a segment named start_up.emacs in Sally's home directory if Emacs is indeed to use it.

A more complex start-up yet does the same things that Sally's does, except that it is expressed in a form that allows for faster, more efficient execution at the time Emacs is started up. If your start-up does many things, you will want to do this to your

start-up as well:

```
 1  ;Nick Romanov's start-up
 2  ;Function definition is used to make it execute faster
 3  ;Petrograd 10/17
 4
 5  (defun Nick-start-up ()
 6        (setq my-personal-name "Nicholas A. Romanov")
 7        (accept-messages)
 8        (opt 'find-file-set-modes 'on))
 9
10  (Nick-start-up)
```

As you know already, lines 1-4 are simple comments, stating what this start-up is and where it came from. They have no meaning to Emacs and are ignored. You will recognize lines 6 and 7 from the earlier examples. Line 6 sets Nick's name for use when he composes mail, and line 7 activates the Emacs message system. Line 9 is blank, and is treated as a comment. Lines 5, 10, and 8 are explained below.

Lines 5 and 10 are interesting ones. Line 5 says, "See the things on lines 6 to 8? They are a set of things to do." Line 8 is the end of the set because of the last close-parentheses on that line, which balances the one at the beginning of line 5. In line 5, the name "Nick-start-up" is given to that set of things to do. That is what defun means, "define function." Thus, "defun Nick-start-up" means that the definition of a function, (i.e., a set of tasks to do) named Nick-start-up begins here. The fact that Nick coalesced all his start-up time tasks allows this "function" to be compiled for faster execution. The open and close parentheses ending the line, (), are necessary and must not be omitted: they mean that "Nick-start-up" has no arguments.

Line 10 of Nick's start-up says, "Do the thing called Nick-start-up" in the same way that line 1 of the first example says, "Do the thing called accept-messages." It says to invoke, execute, or carry out the set of requests and commands that has the name "Nick-start-up," i.e., the set of commands and requests just defined. Why can they not simply be stated instead of assigning this name to them, as was done in the first two examples? You could do this, but the function definition achieves increased efficiency.

Note that lines 6 to 8 are indented, and line up with each other. This is stylistically proper for function definitions; since extra white space is ignored, it is not strictly necessary.

The name "Nick-start-up" is completely arbitrary. You should, however, call your start-up function something like that, except substitute your own name for Nick. The name of the start-up function is not used anywhere except on the line (e.g., line 10 of the last example) that invokes it. It is not the same as "my-personal-name," and is not used by RMAIL.

You may have been wondering about line 8. This is a request in Nick's start-up that invokes the ESC X opt extended request, described in Section 17. The particular line here:

```
(opt 'find-file-set-modes 'on)
```

has the exact same effect as if Nick had typed:

```
ESC X opt find-file-set-modes on CR
```

as soon as he had entered Emacs. The ESC X opt extended request is being used here to invoke the find-file-set-modes option, a very popular and common option that most users familiar with Emacs elect to have on. It causes automatic entry into PL/I mode when find-file (^X^F) is used to read in a PL/I program, FORTRAN mode when a FORTRAN program is read in, and so forth. It is not on by default, since a beginning Emacs user might not know how to use PL/I mode. New users should be able to edit PL/I (or any other language) in Fundamental mode until they acquire proficiency in these special modes.

Any Emacs extended request can be invoked in this way from a start-up. You will note two differences between the way extended requests are issued to Emacs and the way they are stated in start-ups. First, instead of typing an ESC X, and ending with a carriage return, you put the extended request and its arguments in parentheses, as for accept-messages (which had no arguments) in the first example. Second, you put the apostrophe character before all arguments to the extended requests (in this case, the extended request is "opt" and the arguments are "find-file-set-modes" and "on"). This is necessary to differentiate constant arguments (e.g., the keyword "on") from variables (e.g., a variable named "on"). If you leave out the apostrophes, you receive an error about undefined variables, which is in fact what you have specified.

The opt extended request is by far the most common extended request to use in start-ups, other than accept-messages. The full description of all of the options to the opt extended request may be found in Section 17.

Compiling a Start-up

As was mentioned before, start-ups may be compiled, i.e., translated into hardware machine language, to effect faster execution. Emacs start-ups are actually computer programs written in Lisp programming language, a powerful and flexible language, in which Emacs itself is written. More information about Lisp can be found in the Extension Writers' Guide. If you achieve proficiency in extension writing, you can vastly increase the power and sophistication of your start-up as well.

However, to compile an Emacs start-up, all you need to know is how to use the Lisp compiler. This is easy; it is just like any other Multics compiler. You invoke it, giving it a source segment in the Lisp language (an Emacs start-up as described above is such a segment), and it produces an object segment.

To compile an Emacs start-up, this is what you must do:

1. Find out the pathname of the Lisp compiler from a knowledgeable person at your site. If your site has Emacs, it must have the Lisp compiler as well. Its name is lisp_compiler, or lcp for short. It is probably in the same directory as the segment "lisp." If you have used Emacs in your process, use the Multics where command (type: wh lisp) to determine the name of this directory.

2. Prepare the Emacs start-up as described above, with requests, function definitions, and comments in it. Write it out to a segment (file) named "start_up.emacs.lisp" (make sure the name "start_up.emacs" is not on this segment, or you will be in danger of destroying it).

3. Invoke the Lisp compiler:

   ```
   lisp_compiler path
   ```

   where path is the pathname of the start-up.emacs.lisp segment you just prepared; watch out for the underscores (not hyphens) in the name of the Lisp compiler.

4. The compiler may issue diagnostics. Warnings of the form:

   "my-personal-name undeclared -- henceforth assumed special"

   and:

   "(accept-messages default-emacs-start-up opt) -functions referenced but not defined"

   are normal. Any other diagnostic may be an indication of an error. Check your start_up.emacs.lisp and go back to step 2.

5. Assuming compilation was successful, you now have an object segment called start_up.emacs in your working directory. Place it in your home directory, making sure you have read and execute access. You now have a compiled start_up.emacs.

It is not good to let others share your start-up, because of the personal name in it: start-up's are personal. If your associates want to use your start-up, either prepare start-ups for them, or let them copy and edit yours.

MORE FEATURES YOU MIGHT WANT

Below are some other lines you might want to put in your start-up, or take as examples. If you use them, or lines like them, they should be put wherever you put the rest of the requests you invoke at start-up time, either in your start-up function if your start-up is compiled, or standing alone, as in the second example, if your start-up is interpreted (i.e., not compiled). In general, the order is not important.

Here are the sample forms (the correct Lisp term for these requests):

```
(and (eq tty-type 'vip7801)(set-screen-size 10.))
```

This says, "If I am using a (Honeywell) VIP7801 terminal, set my screen size to ten." The tty-type is a variable that contains the terminal type you are using -- the terminal type is just like the name of the terminal in the system's Terminal Type File, except it is all lowercase. The "eq" means "equal, the same as." The set-screen-size extended request is being used here, with an argument of 10. Note the decimal point: all numbers in Lisp are in base eight (octal) unless followed by a decimal point, which puts them in base ten, the base that people usually use for numbers. An apostrophe is not necessary before numeric arguments, but you

can include one. Again, this is the same as if the user had issued the extended request:

ESC-X set-screen-size 10 CR

This form:

```
(and (< ospeed 120.)(create-new-window-and-stay-here))
```

says, "If I am logged in over a communications line operating at less than 120 characters per second (1200 baud), do a ^X3 (create-new-window-and-stay-here), putting me in 2-window mode and leaving me in the first window." This is often what people want to do on low-speed lines, cutting down the amount of printout. Note that the command names (e.g., create-new-window-and-stay-here) associated with a given keystroke (e.g., ^X3), are useful in a start-up.

In the above two examples, "ospeed" and "tty-type" are variables, and it is normal to receive warnings about them from the compiler, which will "Declare them special." These warnings can be ignored.

The form:

```
(set-permanent-key "^H" 'backward-char)
```

is an example of a form that changes a default key binding. The person using this form wants the backspace (control H) key on his or her terminal to go backward a character (do what control B does). In this way, you can switch the assignment of any keys in your customized Emacs environment. The set-permanent-key extended request operates the same as described in Section 15 and creates all-buffer key bindings during the course of an Emacs invocation.

Note that the circumflex and the ^H (not a real control H) are in quotes instead of behind an apostrophe. This is recommended for key names, which can contain special characters like semicolon or parentheses, which otherwise have meaning. When quotes are used, the apostrophe (') is not necessary.

The form:

```
(defprop pl1 electric-pl1-mode suffix-mode)
```

("defprop" is peculiar insofar as no quotes are needed on its arguments) says, "Invoke electric-pl1-mode everytime a segment with a suffix of ".pl1" is read in via ^X^F." Thus, this elects electric PL/I mode as the mode to be entered for all programs.

Similarly:

```
(defprop compin fillon suffix-mode)
```

says invoke fillon, i.e., enter fill mode, whenever a "compin" segment is read in.

APPENDIX H

POP-UP WINDOWS

Emacs provides an option which causes windows to be created and destroyed dynamically as new buffers are switched into and as dedicated buffers are created and destroyed. This option causes new windows to "sprout" on various points of the screen, cutting up or removing old windows, or dynamically reorganizing the screen as new buffers are selected. One goal of this technique is to display simultaneously as many as possible of the things that you were working on recently by packing the screen full.

This option, pop-up windows, is experimental; many Emacs users like it, and many do not. It can be turned on via the "opt" (option) extended request:

ESC X opt pop-up-windows on

and turned off similarly. It is off, by default.

When in pop-up window mode, the standard window-selecting, creating, and destroying requests are used as in non-pop-up window mode. You will find that windows appear less frequently in pop-up mode if there is only one window on the screen to start with (the assumption is that if you have only one window, you are doing that deliberately), so it is often necessary to divide the screen yourself to get pop-up-windows "rolling."

In pop-up window mode, requests that create or switch to a buffer create a new window if the buffer being switched to is not already on the screen. If the buffer is already displayed, these requests switch to the appropriate window. It is virtually impossible to get two windows displaying the same buffer in pop-up window mode.

APPENDIX I

LISTING EMACS TERMINAL TYPES

This appendix contains a description of the list_emacs_ctls command, which lists all known terminal types.

Name: list_emacs_ctls

This command produces a list of all known Emacs terminal types, or verifies the existence in your search rules of specified Emacs terminal controllers.

Usage

list_emacs_ctls {terminal_type}

where terminal type can be a single starname.

Example

When given with no arguments, this command lists all Emacs terminal types:

```
! list_emacs_ctls
  Listing of Emacs terminal controllers:

  in >system_library_unbundled
     aa-ambassador                adds980
     adm2                         adm3a
     ambassador                   ambassador_24l
     ambassador_30l               ambassador_48l
     ambassador_60l               cdc713
     concept100                   delta4000
     dg132b                       dg132b120
     dg132b60                     dku7102
     dm1521                       dm2500
     dm3000                       fox1100
     glasstty                     h19
     hazeltine1510                heath19
     hp2645                       hp2648
     ibm3101                      ibm3101_1x
     ibm3101_2x                   infoton
     infoton100                   iq120
     iriscope200                  ktm2
     micromind                    mmind
     owl1200                      pe550
     printing                     regent200
     smart_ascii                  smarterm
     supdup                       supdup_output
     tdv2220                      tek4023
     tek4025                      tek4027
     teleray1061                  teleray3700
     terditor                     translex
     tvi912                       tvi920
```

list_emacs_ctls

```
tvi950                      umind
video_system                vip7200
vip7201                     vip7205
vip7300                     vip7800
vip7801                     vip7813
vip7814                     vip7823
vis200                      vistar
vt100                       vt100fc
vt100w                      vt100wfc
vt100ws                     vt102
vt102_132c                  vt102_132c_oflow
vt102_80c                   vt102_80c_oflow
vt102_oflow                 vt132
vt132_80c                   vt132_80c_oflow
vt132_oflow                 vt132p
vt132p_80c                  vt132p_80c_oflow
vt132p_oflow                vt52
z19
```

When given with the name of a terminal type as an argument, list_emacs_ctls either verifies the existence of any Emacs terminal controller in your search rules that matches the starname, or prints the message "No Emacs terminal controllers found."

APPENDIX J

OVERWRITE MODE

The Overwrite minor mode is an offshoot of Emacs Fundamental mode that changes the way characters are inserted into the buffer. With Overwrite mode turned on, inserted characters overwrite existing characters, rather than being inserted in their place and pushing them to the right. Some users prefer this minor mode when editing tables of statistics.

To enter overwrite mode, type ESC X and then overwrite-mode. Emacs adjusts the mode line accordingly:

Emacs (Fundamental <overwrite>) - main

To see the results of editing in overwrite mode, let's edit a line without overwrite mode and then with overwrite mode turned on. To fix the line:

The wirk is dune.

in the usual manner, you move the cursor to the letter i in "wirk", delete the i, and insert the letter o. To change "dune" to "done", you would repeat the same procedure to change the u to an o.

In overwrite mode, place the cursor under the i in "wirk" and type an o. The o overwrites the i. Position the cursor under the u in dune and type an o, and again, the letter is overwritten.

When you actually insert and delete characters in this mode rather than merely replacing them, it gets more complicated. To change:

The work is done.

to:

The project is done.

without overwrite-mode, you would delete the word work and insert the word project in its place. Fundamental mode understands that you are replacing a word and keeps the space between the words intact.

With overwrite-mode turned on, however, Fundamental mode reacts differently. If you delete the word work and type in the word project, you end up with:

The projecte.

since, true to its name, it overwrites--regardless of spacing, exactly as you type.

To turn overwrite-mode off, type ESC X overwrite-mode-off.

New Emacs Features

Norman E. Powroz
Department of National Defence
Ottawa, Canada

Paul Benjamin
Honeywell Bull Inc.
Phoeniz, Arizona

September 15, 1988

Abstract

Since Multics release MR11.0, Emacs, the Multics full-screen text editor, has undergone a series of enhancements, and more new features are planned. In MR12.0, the ability to process multi-segment files was added, and Emacs was made more sensitive to external file changes. In MR12.1, new features for program development and compilation were provided. In the upcoming release MR12.2, Emacs will be further upgraded with the addition of support for 8-bit character sets, and the inclusion of a facility to manage vertical windows on certain devices.

This paper describes the implementation of these new features, and discusses ways in which to utilize the new capabilities. The impact on existing user extensions to Emacs will be reviewed, and advice provided on how to avoid problems.

1 Introduction

Emacs, the Multics full-screen text editor, has long been recognized as one of the best pieces of software of its kind. Originally designed to ease the task of textual file manipulation, its ease of use, and extensibility have led to it becoming a mainstay on most Multics systems. As well, its collection of powerful extensions has made Emacs into one of the most significant productivity aids within the Multics environment.

Since its introduction in the early 1970s, Emacs has continually grown in capability. Many internal features have been added, and others improved. Literally dozens of Multics users and developers have built external packages for use with Emacs, and many of these have been incorporated into the standard product offering, thereby increasing both the power, and the attractiveness of Emacs for use either as a development tool, or as a component of a production application. Examples of this type include the various programming language "modes", and some of the features described in later sections of this paper.

2 Multics Release MR12.0 Enhancements

2.1 Multi-Segment File Improvements

The major improvement to Emacs for MR12.0 was the addition of the capability to process multi-segment files (MSFs). The lack of this feature had been a long-standing limitation of Emacs; elimination of the restriction was not a simple problem, however.

Since its initial design, Emacs has been oriented toward the use of a single segment to contain the text or other data being manipulated. Its earlier versions were also not so well-defined in terms of the internal modularity of the software in comparison to the function being performed. Improvements in this area have taken place over the years, with the result that the majority of the work to implement MSF handling could be localized.

Even so, implementation of the feature required that the Lisp program

e_multics_files_ be completely rewritten. The "ripple" effect of this major change meant that any Emacs program which created or modified files might no longer work. Generally speaking, the *external* view of the functions provided within **e_multics_files_** had not changed. However, many programs used the internal lower-level functions of **e_multics_files_**. All of these programs required modification. In most cases, the affected programs were user-written extensions to Emacs, including some of the most popular such as **forum-mode**, and **find-lisp-source**. Unfortunately, the required changes were not necessarily trivial.

Along with the changes to **e_multics_files_**, the Multics system subroutine **msf_manager_** required some minor changes in order to provide the necessary functionality required by the new Emacs file handler. These changes had no impact an any existing programs, and thus could be largely ignored by the Multics user.

2.2 Other File-Handling Improvements

Along with the MSF capabilities added to **e_multics_files_**, improvements in the general file interfaces were also made. These changes have a much less dramatic impact on the average Emacs user, as they do not affect the types of files that Emacs can handle, but rather they build in a few safety factors which prevent inadvertent damage to files.

In previous versions, Emacs was not cognizant of any file activity occurring in other processes, or even in other parts of the same process. As a result, the results of simultaneous update activity could easily be lost. For example, if two users each had write access to a file, and both were attempting to change the file, Emacs would not notice that the file had been modified during the period in which a copy existed in an Emacs buffer. In the end, the work of one user would be lost, as Emacs would simply overwrite the file with its buffer contents.

Another problem was that Emacs would not test for existing files when it attempted to write out a buffer. When the *write-file* command was given, Emacs did just that—it wrote the file, regardless of whether a like-named file already existed. It was therefore extremely easy to overwrite an existing,

possibly very important file, with the contents of the current Emacs buffer. This behaviour often led to a sudden cry of anguish from the novice user, and from many not-so-novice users.

Fortunately, these problems have now been eliminated. Whenever a file is read into a buffer, Emacs fetches the **date_time_contents_modified** field from the directory entry for the file, and stores it in the internal file information structure for the buffer. Whenever the buffer is to be written out, Emacs compares the value of the saved field with the current value of the field in the directory entry for the file. If the two match, Emacs proceeds with the output, and the contents of the buffer are written to the file. If however the two do not match, then Emacs assumes that some external process has modified the file, and it first queries for permission to write to the file, in order to avoid the possible loss of other changes.

The same type of action takes place when a buffer is to be written to a newly-specified file. Emacs first tests for the presence of an existing file with the same name. If none is found, the output proceeds, and a new file is created. If an existing file is found, then Emacs queries for permission to overwrite the file, or to cancel the output request, in order that a new filename may be specified.

2.3 Using the New Features

Making use of the new MSF capability is straightforward. Simply put, any of the standard Emacs commands for file manipulation, such as *find-file*, *write-file*, and *save-same-file* now will accept any MSF as input, or will create an MSF on output, if necessary. It is no longer necessary to manipulate the individual components of an MSF as separate segments; Emacs views the entire MSF as a contiguous whole, in the same manner as any other Multics file manipulation software.

In a similar fashion, using the other new file-handling features is quite straightforward. All of these features are controlled by the Emacs option mechanism, thus permitting users to tailor the settings to individual preferences. Those who are used to living dangerously may continue to do so by the appropriate option settings, and the changes will effectively become

invisible. For those who prefer to err on the side of caution, a simple modification to the emacs start_up will provide all of the protection necessary.

2.4 Avoiding the Pitfalls

On the whole, there are very few problems that can arise when manipulating files using the new capabilities of Emacs, and the services provided by **e_multics_files_**.

First, for the general Emacs user about the only caution necessary pertains to the specific type of MSF to be read or written by Emacs. Standard stream files, as created by other text editors, or by other utilities, such as the *write* request of **forum**, can be read and written without regard to specifics of the internal format. After all, a stream file is a stream file, regardless of what created it.

However, non-stream MSFs abound on the average Multics system. In many cases, these are indexed files created by **vfile_**. To **vfile_**, these files have a specific organization, namely that the first component contains the index, and remaining components contain the data comprising the contents of the file. In Emacs, however, this type of special organization is ignored, as Emacs expects to see only a stream file. As a result, if one reads an indexed **vfile_** MSF, and then writes it back out, the original organization of the file will be destroyed. Emacs will rather nicely convert the file from indexed to stream organization; unfortunately, this can tend to play havoc with the average application as it will no longer be capable of locating its data records at the appropriate places in the file.

The more technically oriented user, especially one who wishes to develop extensions to Emacs, should be aware that the changes in **e_multics_files_** can have an impact on the design of the particular extension. In general, only the well-documented, external interfaces of this program should be used. This type of approach ensures that the extension will continue to work across new versions of Emacs, or that any required forward-fitting will generally be fairly trivial in nature. The danger of using the undocumented internal interfaces of Emacs, as with all other Multics software, is that the Honeywell Bull developers do not guarantee that these interfaces will

remain the same from release to release. They may undergo radical changes in calling sequence or functionality, or they may even disappear entirely in a new release.

3 Release MR12.1 Improvements

3.1 Improved file_output Buffer Handling

With the issue of Multics Release MR12.1, a collection of minor enhancements was added to Emacs, mainly to improve the use of Emacs as a development tool for the compilation and error scanning of programs. These changes all affected the manner in which Emacs dealt with buffers created by various uses of the **file_output** commands. The Emacs commands *compile-buffer* and *comout-command* both use the Multics **file_output** command to accomplish the majority of their work.

3.2 How They Work

The *compile-buffer* command now places any error messages in a buffer named "Compilation Errors" instead of "file-output". This removes the conflict between this command and its cousin, *comout-command.* If the option *one-error-scan-buffer* is turned off, then the error messages will go into a buffer named "*buffer* Errors", where *buffer* is the name of the buffer that had been compiled. This approach allows for the separate compilation and error scanning of multiple programs.

If the option *compile-two-windows* is on, then *compile-buffer* will automatically split the screen, if necessary, putting the error message buffer in the second window. If the option *compile-local-display* is on, and *compile-two-windows* is off, then the error messages are shown as a local display, in addition to being put into a buffer. If neither of these options are set, then a one-line local display gives an indication as to whether the compilation was successful.

If *comout-command* is given a numeric argument, it executes the command

comout-command-to-buffer. This command operates in the same manner as *comout-command*, except that it first prompts for the name of the buffer in which the output should be placed. In this case, the contents of the file-output buffer are not affected.

If the output buffer for the *comout-command* or *comout-command-to-buffer* commands is marked as **read-only**, or if it has an associated pathname and has been modified, then the user is warned when the prompt for the Multics command is issued. If the user aborts the command, the buffer will be left undisturbed; if the user enters a command line, then the command will be executed, the current contents of the buffer will be replaced, the **read-only** flag will be removed, and the pathname will be dropped from the buffer.

While the combination of all of these options seems a bit daunting at first, a little experimentation quickly leads to a preferred mode of operation. After that, a minor change to the user's Emacs start_up, and everything will function as desired.

4 Planned MR12.2 Enhancements

The major enhancements to Emacs in Multics Release MR12.2 were actually developed originally in 1984 and 1985, and were first installed in a version of Emacs operating under Release 10.2. These enhancements enabled Emacs to process 8-bit character sets, and to manage a screen whose windows divided it vertically, rather than the typical horizontal window management used in previous versions of Emacs.

The impetus for these new features came from a requirement of the Canadian Department of National Defence. DND was embarking upon the development of an electronic publishing system, and Emacs was to be used within the system for text editing, and manipulation of raw manuscripts, prior to final composition and formatting. By law, the Department publishes all documentation in the two official languages of Canada, namely English and French. Manipulation of the special characters of the French language, such as *è*, *ç*, and *Á* required the use of an 8-bit expanded character set, as well as specially engineered terminals. In addition, part of the

publication process includes a comparative analysis of the two languages of a document. A side-by-side visual presentation of the two languages is the most natural approach for this type of work, so the Department contracted for the development of such a facility in Emacs.

4.1 Expanded Character Sets

Implementation Details

As part of the development of the 8-bit facility, one other change was required—the Multics Communications System needed a modification to allow it to pass 8-bit characters, as the MCS was only designed for older, 7-bit character sets. This change was installed on the Department's system under Release MR10.2, and was officially released to the rest of the Multics community as part of MR12.0. The change itself is extremely minor, and simply removes a restrictive test.

The Emacs changes themselves were somewhat more major, as parts of the outer PL/1 shell were affected, in addition to the main Lisp internals of Emacs. As it turned out, Emacs was rife with tests to trap and eliminate any character with an octal value of higher than 177. Every one of these tests had to be found, and either be eliminated, or be bounded by additional code to cater to the special case of processing an 8-bit character. Unfortunately, even the Lisp compiler itself contains many checks for characters that do not fit the 7-bit set. As such, special handling is necessary to present 8-bit characters within program code, making the job of program design a little more difficult.

Eliminating simple tests for 8-bit characters was fairly straightforward, however other cases were not quite so easy. For example, a very real requirement existed to allow 8-bit characters to be assigned to Emacs commands. Implementation of this feature alone required modifications to the character parsing routines, and the key-binding functions, as well as a new method of representing the assigned character string in error and help displays. The designation *ext-* is used to indicate such characters in key-bindings, thereby making it simpler than having to specify the octal value of the character.

Many of the 8-bit character tests existed in the PL/1 shell, especially in areas dealing with MCS negotiation to establish proper echo handling, breaktables, and communications mode setting. Expansion of the breaktables was necessary, in conjunction with the MCS change mentioned previously, as both the MCS and Emacs otherwise treated an 8-bit character as a break indicator to be mercilessly thrown away, rather than incorporating it into the incoming text stream. Synchronization of this activity is necessary, as the use of an 8-bit Emacs with a 7-bit MCS causes an immediate FNP crash. Conversely, the use of a 7-bit version of Emacs with an 8-bit MCS is quite acceptable, as many sites are now unknowingly doing.

Although an ISO standard for 8-bit character sets exists, it is fairly recent, and not all terminals follow the standard. As a result, the specific mapping of a certain pattern of eight bits to a specific displayable character is therefore left to the imagination of the terminal designer. To avoid this problem, Emacs makes no assumptions about the pairings of 8-bit values to displayable characters. Its only assumptions remain those assigned to 7-bit characters; this is a safe approach as the 7-bit standard has been in existence for a number of years, and is slavishly followed by all manufacturers of ASCII-based communications devices.

Specification of the specific pairings for a given terminal can easily be accomplished via the the mechanism of the Emacs Terminal Controller, known as the CTL. As each different type of terminal requires a unique CTL module to define the display control sequences for the terminal, specification of additional characters can easily be built into the CTL, to be executed every time that Emacs initializes the terminal. As well, a new global variable, *DCTL-extended-ascii*, was defined for use by the CTL. Defining the variable as *true* specifies that the terminal is capable of supporting 8-bit characters. The addition of a table defining the precise pairings within the CTL completes the necessary interfaces to individual terminal devices. This approach also caters to national character requirements that may not fit within the ISO 8-bit standard, without the need to replace characters from other portions of the standard 7-bit character set definition.

4.2 Vertical Window Management

The requirement for a vertical window capability in Emacs came as a natural extension of the standard used for the presentation of bilingual material in Canadian government publications. Typically, bilingual documents are formatted using a two-column side-by-side layout, with English text in one column, and equivalent French text in the other. As this approach eases the comparison of the two languages, it was desirable to carry over the same capability into the text-editing software.

A contract was let to Honeywell Canada for the development of the vertical window management capability, as well as a series of support functions which would use the new features within the Department's publishing system.

Implementation Details

In order to implement this facility, the concept of the *split* was defined, thus creating a new object for Emacs to handle. As Emacs already knew about such global objects as buffers and windows, the same type of approach could be applied to the *split*. Basically, a *split* has most of the same properties as a window. If it is not on display, then it does not exist. While it is on display, it contains a buffer of information in the same manner as a normal Emacs window. The major difference between a split and a window is the orientation of the object when it is on display, as a window represents a horizontal division of the screen while a split represents a vertical division.

One other difference between the two is not as immediately obvious. Emacs manages and displays windows with no external assistance from the hardware of the terminal in use. All decisions concerning the placement of text within one or more windows are made by Emacs itself, as are all recalculations concerning the repainting of the screen to ensure that the content of each individual window is consistent at all times. However, in the case of a split, Emacs requires that the terminal in use be capable of differentiating among the separate splits. Effectively, the terminal must act as if each split is an independent screen, such that update activity in one split has no impact on the display contained in another active split. In this manner,

each split can be managed independently without the need to worry about interference with another displayed split. This approach lessens the amount of work needed by the Emacs Window Manager, as it can safely assume that update activity in one split requires no adjustment of any other split on display. Conversely, the Window Manager must constantly be aware of adjacent window intrusion whenever a normal window is modified.

Admittedly, while an approach that uses terminal hardware is simpler to implement, it does lessen the transferability of the overall functionality. The contract between the Department and Honeywell did not specify the manner in which the functionality was to be achieved; this decision was left to the Honeywell development staff.

As with the 8-bit character capability, vertical window management is activated via functions contained within the CTL module. A second global variable, *DCTL-hardware-windows-availablep*, is used to indicate to Emacs that the particular terminal is capable of supporting split-screen displays. In addition, the CTL must contain a function which will instruct the terminal to create a new split. This function, called **DCTL-create-split**, must accept the split number, home column and line position of the split on the screen, and the width of the split in columns and lines. It must then emit the necessary control characters to the terminal to cause creation of the split.

Emacs extensions can then take advantage of this new display capability, simply by calling the appropriate entrypoints within Emacs. Buffers can then be displayed in splits or windows, with Emacs providing the necessary management of each. One cautionary note, however is that the Emacs minibuffer should be placed in a split that spans the entire width of the screen. As the minibuffer cannot be scrolled to handle overlength lines, a split that is too small would mean that much of the minibuffer's contents would not be on view. This situation can make some commands a little frustrating to use.

Within the Department of National Defence publishing system, Emacs extensions have been developed to utilize the vertical display facility, primarily for the visual comparison of two files in different languages. This same type of display is also extremely useful as a programming tool, as two ver-

sions of a program can be easily be compared for consistency, without the need to modify vertical reference points as occurs with "over and under" windows.

5 Conclusion

The Emacs refinements offered in Multics Releases 12.0 and 12.1 offer new functionality for the full range of Emacs users, and remove a long-standing limitation on the size of a file that Emacs could handle. With today's larger applications, enormous source files are in common use, so elimination of the single segment limit is highly desirable in order to expand the range of available text editors, and provide more packaging options for the source code files.

The soon to be released extensions for MR12.2 remove one more limitation, and also offer the Emacs user with a new flexibility in screen display management. Restrictions on character sets have plagued non-English users of Multics since its inception. The 8-bit capability now provides the full range of displayable characters offered by most commercially available terminals, removing the need for character set juggling that had been the only available option in the past. As well, the new vertical window management facility extends the range of possible screen display configurations, and makes Emacs even more attractive for use as part of an application, or as a text-entry or programming tool.

While the MR12.2 extensions will not be generally available to Multics users for a few months, they have been in use within the Department of National Defence for three years, and have been extremely stable during that time. A few annoyances and troublesome side effects were discovered and eliminated within a short time after initial delivery, but since then the modifications have proven trouble-free.

INDEX

MISCELLANEOUS

E

F

W

Y

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS EMACS TEXT EDITOR USER'S GUIDE ADDENDUM F

ORDER NO. CH27-00F

DATED NOVEMBER 1986

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

CUT ALONG LINE

CUT ALONG LINE

FOLD ALONG LINE

PLEASE FOLD AND TAPE–
NOTE: U.S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA 02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

MULTICS EMACS TEXT EDITOR USER'S GUIDE ADDENDUM F

SUBJECT

Changes to the Manual

SPECIAL INSTRUCTIONS

This is the sixth addendum to CH27, Revision 0, dated December 1979. Refer to the Preface for "Significant Changes."

Insert the attached pages into the manual according to the collating instructions on the back of this cover. Throughout the manual, change bars in the margins indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated in the next revision of this manual.

Note: Insert this cover after the manual cover to indicate the updating of the document with Addendum F.

SOFTWARE SUPPORTED

Multics Software Release 12.0

ORDER NUMBER

CH27-00F

November 1986

46226
5C986
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
iii through vi	iii, iv v, blank
Section 5	Section 5
12-3, 12-4	12-3, 12-4
14-1, 14-2	14-1, 14-2
15-5, 15-6 15-6.1, blank	15-5, 15-6 15-6.1, blank
17-19, 17-20 17-20.1, blank	17-19, 17-20 17-20.1, 17-20.2
17-47, 17-48 17-48.1, 17-48.2 17-49, blank 17-49.1, 17-50	17-47, 17-48 17-49, blank 17-49.1, 17-49.2 17-49.3, 17-50
C-15, C-16	C-15, C-16
Appendix I	Appendix I

The information and specifications in this document are subject to change without notice. Consult your Honeywell Marketing Representative for product or service availability.

©Honeywell Information Systems Inc., 1986
File No.: 1L33

11/86
CH27-00F

Together, we can find the answers.

Honeywell

Honeywell Information Systems
U.S.A.: 200 Smith St., MS 486, Waltham, MA 02154
Canada: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
U.K.: Great West Rd., Brentford, Middlesex TW8 9DH **Italy**: 32 Via Pirelli, 20124 Milano
Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F. **Japan**: 2-2 Kanda Jimbo-cho, Chiyoda-ku, Tokyo
Australia: 124 Walker St., North Sydney, N.S.W. 2060 **S.E. Asia**: Mandarin Plaza, Tsimshatsui East, H.K.

40367, 4C584, Printed in U.S.A.

CH27-00