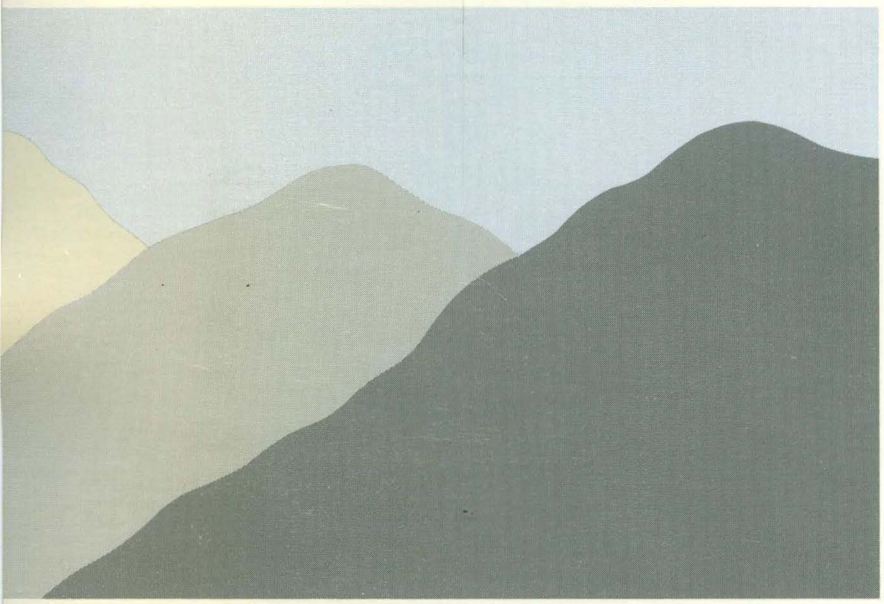

ROS Text Editing Guide

RIDGE

9051

ROS Text Editing Guide

Ridge Computers

Santa Clara, CA

first edition: 9051 (SEP 84)

© Copyright 1984, Ridge Computers.
All rights reserved.
Printed in the U.S.A.

PUBLICATION HISTORY

Manual Title: ROS Text Editing Guide

first editon: 9051 (SEP 84)

NOTICE

No part of this document may be translated, reproduced, or copied in any form or by any means without the written permission of Ridge Computers.

The information contained in this document is subject to change without notice. Ridge Computers shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the use of this material.

ACKNOWLEDGEMENT

This software and documentation is based in part on the fourth Berkeley Software Distribution, under license from the regents of the University of California. We acknowledge the following individuals for their part in its development: Ken Arnold, Rick Blau, Earl Cohen, Robert Corbett, John Foderaro, Mark Horton, Bill Joy, Randy King, Jim Kleckner, Steve Levine, Colin McMaster, Geoffrey Peck, Rob Pike, Eric Scott, and Eric Shienbrood.

These tutorial guides are based on documents created at Bell Laboratories to describe UNIX System software, and at the University of California, Berkeley, to describe bsd 4.2. Credits are given on the first page of each document contained in this volume.

Some text has been changed to more accurately describe Ridge Computers' implementation of the software. Inappropriate material may have been deleted.

UNIX is a trademark of Bell Laboratories.

PREFACE

The ROS Text Editing Guide (manual 9051) is a collection of tutorial documents on text-editing software available with the Ridge Operating System. These sections supplement the editor reference pages in the ROS Reference Manual (9010).

TABLE OF CONTENTS

Tab Label		*ROS Reference Manual Page*
REDIT	- Ridge Editor	redit(1)
ED	- ED editor	ed(1)
Advanced ED	- more on ED	ed(1)
EX ref	- EX editor reference section	ex(1)
EDIT	- EDIT editor	edit(1)
EX/EDIT	- EX and EDIT combined reference	ex(1), edit(1)
VI	- VI editor introduction	vi(1)
VI ref	- VI editor reference section	vi(1)
SED	- Stream Editor	sed(1)

REDIT — Ridge Text Editor Reference Manual

CHAPTER I INTRODUCTION

This manual documents **redit**, the Ridge text editor. An explanation of each of the commands and operations is given, as well as a short tutorial. This manual also explains differences encountered between the Ridge Monochrome Display and the Text Terminal (Televideo 950). After having read this document, the reader should be able to create and edit files using redit, using either type of terminal.

No knowledge of the Ridge 32 system on the reader's part is assumed beyond a minimal acquaintance with the operating system. The reader is referred to the *Ridge Operating System (ROS) Reference Manual* for such basic operating system information as particulars of file and directory structure, and allocation information for discs.

OVERVIEW

The Ridge text editor, **redit**, is a full-screen, strikeover editor that allows the user to create and modify text files. This chapter briefly describes redit and discusses file structure and types, display formats, and selected blocks and windows.

Redit combines the attributes of a screen- and a line-oriented editor in such a way that the user can define and modify entire blocks of text while retaining control over individual line operations. Screen-oriented attributes allow such features as windows and powerful block commands. The combination of these features allows the user simultaneous access into different areas of a file or multiple files (Figure 1). Line-oriented attributes allow the user to perform operations such as insert/delete line or character, or search-and-replace with various options, as well as to build user-definable command files that execute complex/repetitive editing functions.

Redit is not a formatting program; post-processing (with nroff, for example) is necessary to format output. Nor is redit a multiple mode editor; the editor is always in strikeover mode. Changes or alterations are made by overwriting text directly on the display.

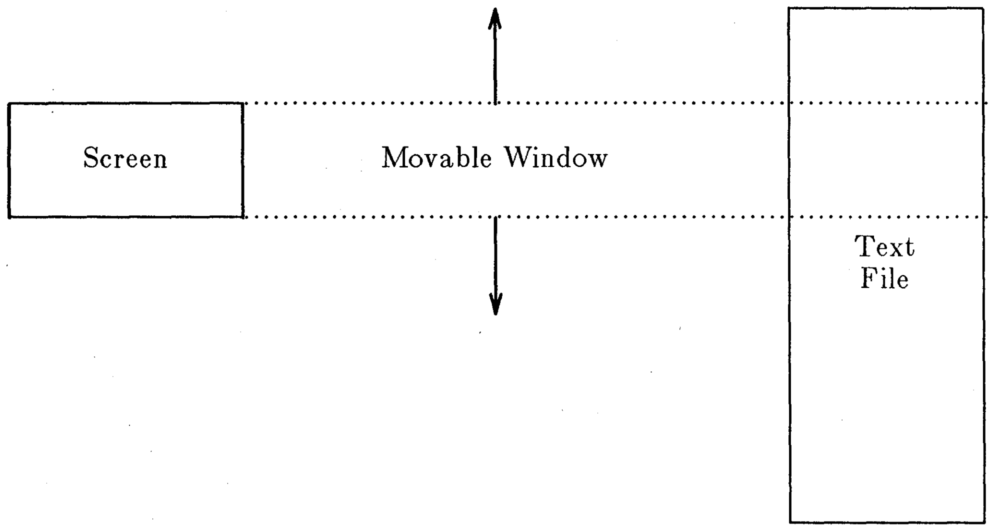

Figure 1. Windows on Text File

Editing is performed using function and other editing keys, and through commands entered on the command line. (Virtually all keyboard keys function normally with redit--e.g., backspace moves one character left.)

Editing capabilities are thus broken into two groups:

• EDITOR COMMANDS	With this group, the user enters the command into the command entry area (top line) of the screen.
• PREDEFINED OPERATIONS	These operations are invoked directly by function keys. The predefined operations have been implemented using various combinations of the editing commands, and represent the most commonly used editing tasks.

GENERAL DESCRIPTION

This section describes the logical structure of text files, ASCII and work files, screen formats, and the concept of selected blocks and windows.

Logical Structure of Text Files

A text file consists of a set of records referred to as lines. Each line has a length attribute; the length describes the position of the last non-blank character in the line. The length of a line may range from zero (0) to 128 characters. A length of zero denotes a blank line.

Each line can be identified by its position relative to the beginning of the file. The position (line number) has a value in the range of 0 to 99,999,999, inclusive. Lines are effectively renumbered if preceding lines are inserted or deleted.

ASCII Files and Work Files

"ASCII file" refers to a file that is directly readable by such Ridge subsystems as the editor, language compilers, and utility programs. When an existing ASCII file is to be edited, the file is first copied to a "work file." The work file provides rapid, random access to any place in the text; this file is not directly readable by other Ridge subsystems. During the editing session, changes to the text are made only to the work file, not to the ASCII file. At the end of the editing session, the text portion of the work file is copied to a new ASCII file.

The work file is saved by default between editing sessions, which provides three benefits: time spent copying the ASCII file to the text file at the beginning of the editing session is eliminated, some contextual information (such as current position in the file and tab settings) is maintained across editing sessions, and the work file is an additional backup to the ASCII file. (Chapter 3 explains how to save a "lost" work file. In the user's directory, the work file bears a ".e" extension to the original ASCII file name. So, for an ASCII source file named "prime.s", a work file named "prime.s.e" is created.

Screen Format

The Ridge system supports two terminals: the Ridge Monochrome Display and the Text Terminal (Televideo TVI-950C). The Monochrome Display is bit-mapped and a range of font sizes have been developed for it; consequently, the screen format varies according to font size. While the Monochrome Display does not display exactly 72 increments per inch, the font sizes are close approximations to those of typographic fonts.

Figure 2 shows the screen format for the Monochrome Display and Figure 3 that of the Text Terminal.

```
              1.................30 ........................................128
               +------------------------------------------------------------//----+
1 <adv/cmd>   |]<advisory>         :[<command input area>                          |
2 <hdr/sel>   |]<fname>  @  <tline>         | <sfname>   <stline..> <sbline>       |
3 <text1>     |[                                                                   |
4 <text2>     |                                                                    |
5 <text3>     |                                                                    |
49 <text47>   |                                                                    |
50 <cmd>      |S1:Select S2:Clear S3:Copy S4:Move S5:Break S6:Fill S7:XeqCmd S8:Exit
               +------------------------------------------------------------//----+
```

Figure 2. Monochrome Display Screen Format

```
                1................30 ........................................80
                +-----------------------------------------------------------//----+
1 <adv/cmd>  |]<advisory>        :[<command input area>                           |
2 <hdr/sel>  |]<fname>  @  <tline>        | <sfname>   <stline..> <sbline>        |
3 <text1>    |[                                                                   |
4 <text2>    |                                                                    |
5 <text3>    |                                                                    |
24 <text22>  |                                                                    |
25 <cmd>     |1:Prev 2:Next 3:Sel 4:Copy 5:Brk 6:Fill 7:Undo 8:A 9:B 10:C 11:Cmd|
             +-----------------------------------------------------------//----+
```

Figure 3. Text Terminal Screen Format

In these illustrations, the following notation is used:

]	Denotes the beginning of an area into which the user cannot enter text (a protected area).
[	Denotes the beginning of an area into which the user can enter text (an unprotected area).
<adv/cmd>	Advisory/command area. <advisory> is where error messages are displayed; <command input area> is where editor commands can be entered.
<hdr/sel>	Window header and selection status area. The window header consists of three parts:

<fname>	The name of the file visible in the window.
@	The currency indicator which is displayed if the window is current.
<tline>	The line number of the line at the top of the window.

indent The selection status area displays the state of the currently selected block of text (see the SELECT command). The selection status area consists of three parts:

<sfname>	The name of the file containing the currently selected block.
<stline>	The line number of the top line of the currently selected block. (On the Monochrome Display, <stline> is used to indicate both the top and bottom lines of a selected block for sizes 16 and above because the screen isn't wide enough to accommodate the <sbline> field for these larger fonts.)
<sbline>	The line number of the bottom line of the currently selected block.

<text1..n> Area of the screen in which text is displayed. The cursor may be placed anywhere within this block for editing; entering text over existing text automatically replaces that text.

<cmd> Shows the editing functions of the keys above the top row of the keyboard, from left to right.

Selected Blocks

In redit, blocks of text can be defined and manipulated using several of the predefined operations and editing commands. Essentially, the SELECT command (or predefined operation) is used to define a block--to set its bounds (top and bottom lines). Simple keystroke commands can then be used to copy, move, or delete blocks. Editor commands and predefined functions REPEAT command) can be used locally within a block, thus limiting their actions to just a part of the text file (see the REPEAT command). The CLEAR SELECT command nullifies the selection of the current block so another can be selected.

Used in conjunction with the WINDOW capabilities, blocks of text can be passed from file to file, edited, revised, reinserted, etc. Information about the currently selected block--the file name in which the current block resides, and its bounds--is displayed in the selection status area of the screen. (See Figures 2 and 3.)

Windows

Redit offers three scrollable windows (independent, predefined screen areas which act as mini-screens) in which text can appear simultaneously. Up to three text files can thus be displayed and edited without having to close or save any one of them first.

Window A occupies the entire screen area until window B is created. Then, the screen is split horizontally between the two. If window C is created, the B window area is then split horizontally between these two. Figure 4 shows the screen when all three windows have been opened. Movement from window to window is easily accomplished via the WINDOW keys. The sign "@" indicates which window the user is currently working in.

```
 -----------------------------------------------------------
|  -------------------------------------------------------   |
| | ]<advisory>   :[<command input area>                  |  |
| | ]<fname> @ <tline>     | <sfname> <stline> <sbline>   |  |
| | [                                                     |  |
| |                                                       |  |
| |                                                       |  |
| |                      window A                         |  |
| |                                                       |  |
| |                                                       |  |
| |                                                       |  |
| |-------------------------------------------------------|  |
| | ]<fname> @ <tline>      | <sfname> <stline> <sbline>  |  |
| |                                                       |  |
| |                      window B                         |  |
| |                                                       |  |
| |-------------------------------------------------------|  |
| | ]<fname> @ <tline>      | <sfname> <stline> <sbline>  |  |
| |                                                       |  |
| |                      window C                         |  |
| |_______________________________________________________|  |
|___________________________________________________________|
```

Figure 4. Screen Window Format

CHAPTER 2
EDITOR COMMANDS AND PREDEFINED OPERATIONS

This chapter defines the editor commands that are entered in the command entry area. The commands are used singly and in combination with one another to provide extensive editing capabilities.

Syntax Notation

Required keyword characters are indicated in uppercase bold; required arguments are indicated in bold; all punctuation that is shown is required (except commas used to separate a series of options).

Arguments are enclosed by "<" and ">". Options are enclosed by "[" and "]". If, within a group of options, one must be selected, the group is surrounded by "{" and "}"; where a default exists, it is underlined. Lists are indicated by an ellipsis, "...".

The text editor accepts either uppercase or lowercase letters when commands are typed in. Commands must normally be separated from the other elements by one or more spaces; otherwise spacing is unimportant.

Range and Line Specification

<range> is used to specify a block of text to be operated on by an editor command or sequence of commands. <line-spec> is used to specify a single line. The following shows the syntax for both.

<range> =	[(**<filename>**) **.**] **<line-spec>** **..** **<line-spec>**
<line-spec> =	{ **<line-spec,@,B,T,F,L>** } [<{+,-} offset>] { [F,L,N,P] <**'string',"string"**> [I,W,B] }

Where:

@	Indicates the current line.
B	Bottom line of selected block.
T	Top line of selected block.
F	First occurrence or line.
L	Last occurrence or line.
N	Next occurrence or line.
P	Previous occurrence or line.
B	Both upper and lower case are found.
W	Word only is found (not embedded string).
E	Exact character string is found.
<offset>	May be an integer, or the keyword, PAGELEN. "Pagelen" is the number of lines that can be displayed simultaneously.

ATTACH Command

ATTACH <filename> [, **NEW**] [, { **IA, B, C** }]

This command places a file in one of the editing windows (A, B, or C). If the named file does not currently exist, the NEW option creates it. Normally, this is the first command executed after entering the editor, but ATTACH is automatically invoked if a file name is used in calling up the editor itself (see Chapter 4 for details).

Example: The following sequence opens an existing file and attaches it to window B.

ATTACH myfile.s, B

Bugs: In using the NEW option, no spaces may be inserted between the file name and NEW--that is, the sequence must look like this:

ATTACH filename,new

CLEAR Command

CLEAR { **ISELECT, TEMP** }

CLEAR SELECT clears the currently selected block. CLEAR TEMP clears the temporary text stack.

CLOSE Command

CLOSE <filename> [, [NO] { *ASCII* , *SAVE* }]

This command detaches the specified file from any window(s) to which it is attached, and closes the file, but leaves the editor still active. If the ASCII option is specified (by default), and the text has been modified since the last time the text file was copied, the work file (".e") is copied to the ASCII file. If NO ASCII is selected, the work file is not copied to the ASCII file.

If the SAVE option is specified (again, by default), the work file is closed. With the NO SAVE option, the work file is purged after its contents are copied to the ASCII file. NO ASCII and NO SAVE cannot be specified simultaneously.

COPY Command

COPY <range>

This command inserts a copy of the specified line(s) just before the current line (the line the cursor is positioned on).

Example: The following command copies lines 1 through 50 from an open file back into itself.

COPY 1..50

Example: The following command positions a copy of lines 1 through 50 of an open file (named "kit") before the current line of the file in the current window.

COPY (kit). 1..50

DELETE Command

DELETE

This command deletes the current line. (The RESTORE key does not restore a line so eliminated.)

DETACH Command

DETACH { A, B, C }

DETACH removes from a specified window, but does not close, a file.

EXIT Command

EXIT

This command terminates the editor. The equivalent of a CLOSE <filename>, ASCII, SAVE is done for all currently open files.

FIND Command

{ **FIND, F** } <line-spec>

This command locates a line or string within the text file and makes that the location of the current line. In general, <line-spec> can be a line number (e.g., 10), a quoted string (e.g., "String"), or an occurrence (e.g., N = next, P = previous). If a line number is specified, the cursor is placed within the specified line at the same column position it previously occupied. If, however, a string is specified, the cursor is positioned on the first character of the string. Specifying an occurrence by itself is equivalent to specifying the line number of that occurrence. Specifying an occurrence in combination with a string locates that string relative to the start (F), end (L), or current position (P or N) in the file.

FIND only searches forward from the line the cursor is currently on; to search an entire file quickly, "FIND 1" (go to first line), then use FIND to look for the desired occurrence.

Example: The following command makes the previous line the current line.

FIND p

Example: The following command finds the next occurrence of the string "cat", using the word only option.

F n "abc" w

INSERT Command

INSERT [<quoted-string>]

INSERT by itself inserts a blank line prior to the current line. With the <quoted-string> option, the specified string is inserted as a new line prior to the current line. The newly inserted line becomes the current line.

Example: The following command inserts the words "Follow these directions." on a line before the present current line.

INSERT "Follow these directions."

MOVE Command

MOVE <range>

This command inserts a copy of the line(s) specified by <range> just before the current line (the line the cursor is positioned on), and then deletes the original lines.

Example: The following command puts a copy of lines 1 through 50 before the current line and eliminates the original of the lines.

MOVE 1..50

Example: The following command uses the first occurrence of a string to locate a 10-line block, which is then moved from an open file named "kit," to before the current line in the current window. The original lines are deleted from "kit."

MOVE (kit). f "Sec.1" .. f "Sec.1" +10

POP Command

POP

The user can store up to 21 lines of text in a temporary storage stack (the same stack to which the line-delete key sends deleted lines). This command lets the user remove lines from this temporary stack, if the stack is empty, no action takes place.

Use of POP also restores lines into the text that were deleted with the line-delete key, but not the DELETE command.

The stack is a last-in-first-out storage device. The most recent line to be put on the stack, via PUSH or the line-delete key, is the first to be POPped off the stack.

PUSH Command

PUSH

The user can store up to 21 lines of text in a temporary storage stack. It is the same stack to which the line-delete key sends deleted lines.

To copy a line of text to the stack, set the cursor on the line and use the PUSH command on the REDIT command line. A copy of the line will be put on the stack (the original line will stay in place).

The difference between the line-delete key and PUSH is that PUSH does not erase the line from the text. Both features put a copy of the line on the stack.

REPEAT Command

REPEAT (<command> [;<command>] ...)
[, {[IN <range>FR], [COUNT <num>]} , [QUIET]]

REPEAT is used to execute one or more commands repeatedly within an implied or designated range, or until a specified number of iterations are completed.

IN <range> allows iterations to occur within a defined range (a specified group of text lines). If IN <range> is omitted, a range of "1" is implied. If IN <range> is specified, the first line of the range is made the current line prior to the first execution of the command list. If a range of one (e.g., a range of 10..10) is defined, the command list executes one time. If there are no commands that cause the current line to exceed the bounds of the implied or specified range, the REPEAT sequence will continue indefinitely.

The COUNT <num> parameter specifies a limit on the number of times the command list can be repeated.

The QUIET parameter suppresses display update until the REPEAT command has finished iterating.

Example: The following command list finds all occurrences of "abc" and deletes the lines on which they occur. (Note that the EXECUTE/ ENTER key must be pressed to go from one "FIND-REPLACE" sequence to the next. However, COUNT -1 can be appended to make this automatic. If "n" is removed, COUNT -1 acts globally.)

REPEAT (FIND n 'abc' ; DELETE)

Example: The following searches for all occurrences within the currently selected block of "tadpole" and replaces them with "frog."

REPEAT (f 'tadpole';replace 'frog'), IN B..T COUNT -1

REPLACE Command

REPLACE <quoted-string>

This command replaces the currently selected text with <quoted-string>. The text can be explicitly replaced by the execution of a FIND command used in conjunction with REPLACE. Or text can be implicitly replaced based on the current positon of the cursor: if the cursor is currently positioned on a string that has just been located by a FIND string search, that string is replaced by the quoted string. Otherwise, <quoted-string> is inserted into the current line beginning at the cursor position. (See REPEAT for an example of using REPLACE.)

SELECT Command

SELECT

This command uses the current line (where the cursor is positioned) to define the bounds (top and bottom lines) of a text block, which then becomes the currently selected block.

If no block is currently defined, SELECT defines a block of one line. If the cursor is moved to a line outside the current bounds, SELECT redefines the block (its first or last bound is changed). Scrolling up (towards the beginning of the file) causes the top line to be changed; scrolling down (towards the end of the file) causes the bottom line to be changed. If the cursor is moved to a line within the defined block, pressing SELECT does not redefine the current block bounds.

If the current line is not in the same window as the currently selected block, SELECT nullifies the current selection and sets the bounds for a one-line block.

SET TAB Command

SET TAB <col> [**<col>**] ...

This command can be used to override the default display tabs (which are set every eight spaces). A maximum of 15 tabs can be defined. Since this is a two word command, and since a series of numbers may be specified, spacing is significant.

Example: The following command sets up three tabs. The first ranges from column 1 to 10, the next from 10 to 20, the last from 20 to 40.

SET TAB 10 20 40

WINDOW Command

WINDOW { A, B, C }

The WINDOW command selects the current window for editing. The selected window must have a file attached to it.

XEQ Command

XEQ <filename>

This command is used to execute a series of editor commands that are contained in an ASCII file.

Example: Assume a file ("tree") in window A is being edited. Then, to create and edit an XEQ file named "sap" that will open an existing file "leaf" in window B, insert a text string at a specified line in "leaf," and then go back to a specified line in "tree," the following sequence could be used. First, the file "sap" is created. It contains:

```
ATTACH leaf,b
FIND 4
INSERT 'maple sugar'
WINDOW A
FIND 1
```

Next, "tree" is attached to window A. Then the command sequence:

```
XEQ sap
```

causes the desired search, string insertion, and return to line 1 in "tree."

PREDEFINED OPERATIONS

This section describes the predefined operations for the Monochrome Display and the Text Terminal. These operations are invoked by pressing one of the terminal editing keys or one of the functions keys, either unshifted or in combination with the SHIFT key. The predefined operations have been implemented using combinations of the editor commands themselves.

On the Monochrome Display, the most frequently used predefined operations appear on the function keys at the top of the keyboard; less frequently used operations are accessed from the keypads to the left and right of the main keyboard. The exceptions to this are the cursor keys, which are grouped together. (See Figure 5 for an illustration of the Monochrome Display keyboard.) In the Monochrome Display, function keys F1 through F9 have been assigned editor operations; keys F10 through F15 are unassigned. On the Text Terminal, all available function keys (F1 through F11) have been assigned editor operations.

The following descriptions of the predefined operations are arranged alphabetically. Figure 5 represents the keyboard for the Ridge Display, and Table 3 shows which keys on the Monochrome Display and the Text Terminal correspond with which operations, as well as providing the key sequences the terminals will also accept for some operations.

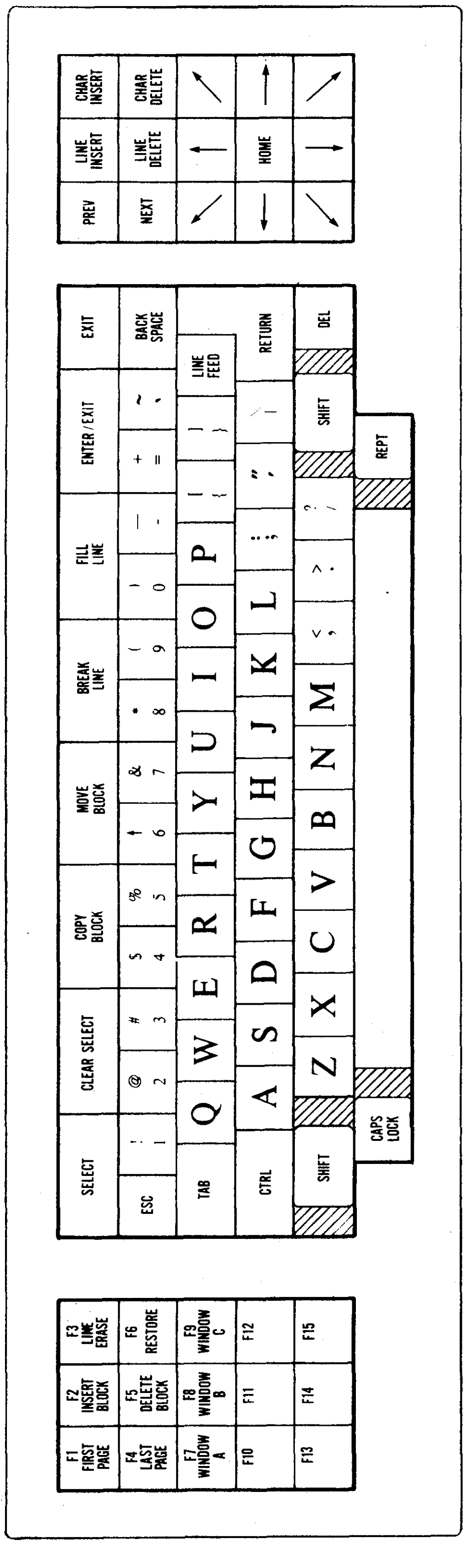

Figure 5. Monochrome Display Keyboard

Table 3. Predefined Operations

Predefined Operation	Text Terminal Key	Monochrome Display Key	Alternate Key Sequence	ANSI Terminal Key Sequence
Backspace	Backspace	Backspace	ctrl-H	ctrl-H
Break Line	F5	Break Line	esc T	esc b
Char Delete	Char Delete	Char Delete	esc W	ctrl-W
Char Insert	Char Insert	Char Insert	esc Q	ctrl-A
Clear Select	F3 Shifted	Clear Select	esc S	
Copy Block	F4 Unshifted	Copy Block	ctrl-AC	esc c
Cursor Diagonal, Left Down		↙		
Cursor Diagonal, Left Up		↖		
Cursor Diagonal, Right Down		↘		
Cursor Diagonal, Right Up		↗		
Cursor Down	↓	↓	ctrl-J or V	ctrl-J
Cursor Home	Home	Home	ctrl	ctrl
Cursor Left	←	←	ctrl-H	ctrl-H
Cursor Right	→	→	ctrl-L	ctrl-L
Cursor Up	↑	↑	ctrl-K	ctrl-K
Delete Block	Line Del Shft	Delete Block (F5)	esc D	
Enter/Execute	F11 Unshifted	Enter/Execute	ctrl-A J	cr* esc x
Exit	F11 Shifted	Exit		ctrl-A j
Fill Line	F6	Fill		esc j
First Page	F1 Shifted	First Page (F1)	?ctrl-A	cr esc f
Insert Block	Line Ins Shft	Insert Block (F2)		esc I
Line Delete	Line Delete	Line Delete	esc R	esc d
Line Erase	Line Erase	Line Erase(F3)		ctrl-E
Line Insert	Line Insert	Line Insert	esc E	esc i
Last Page	F2 Shifted	Last Page (F4)		esc l
Move Block	F4 Shifted	Move Block	ctrl-A c	cr esc m
Next Page	F2 Unshifted	Next	ctrl-A A	cr esc n
Previous Page	F1 Unshifted	Prev	ctrl-A @	cr esc p
Restore (Undo)(1)	F7 Unshifted	Restore (F6)		esc u
Restore (Undo) (all)				esc U
Return	Return	Return	ctrl-M	ctrl-M
Select	F3 Unshifted	Select	ctrl-A B	cr esc s
Tab	Tab	Tab	ctrl-I	ctrl-I
Window A	F8	Window A (F7)	ctrl-A G	esc 1
Window B	F9	Window B (F8)	ctrl-A H	esc 2
Window C	F10	Window C (F9)		esc 3

*carriage return

BACKSPACE

This operation moves the cursor back one position. If the cursor is in column 1, it is moved to the last column of the previous line.

BREAK LINE

This operation splits the current line at the cursor position, creating a new line containing the text trailing the cursor.

CHARACTER DELETE

This operation deletes the character at the cursor position. Text to the right of the cursor on the same line is moved to the left.

CHARACTER INSERT

This operation inserts a space just before the cursor position. Text to the right of the cursor on the same line is moved to the right.

CLEAR SELECT

This operation performs the same function as the editing command, CLEAR SELECT. It nullifies the currently selected block.

COPY BLOCK

This operation inserts a copy of the currently selected block just before the current line, and then nullifies the current selection.

CURSOR DIAGONAL LEFT, DOWN

This operation moves the cursor diagonally, down to the next line and left one column.

CURSOR DIAGONAL LEFT, UP

This operation moves the cursor diagonally, up one row and left one column.

CURSOR DIAGONAL RIGHT, DOWN

This operation moves the cursor diagonally, down one row and right one column.

CURSOR DIAGONAL RIGHT, UP

This operation moves the cursor diagonally, up one row and right one column.

CURSOR DOWN

This operation moves the cursor down one line.

CURSOR LEFT

This operation is the same as a backspace.

CURSOR RIGHT

This operation moves the cursor to the right one column. If the cursor is in the last screen column, it is moved to column 1 of the next line.

CURSOR UP

This operation moves the cursor up one column.

DELETE BLOCK

This operation deletes the lines in the currently selected block, then nullifies the current selection.

ENTER/EXECUTE

This operation positions the cursor in the command input area. If the cursor is already in the command input area, pressing the key becomes a signal to interpret the command line for execution.

To NOT execute a command sequence that has been entered into the command input area, and assuming that the cursor is positioned in the area, press the appropriate window key to move the cursor to the desired current window.

EXIT

This operation performs the same function as the EXIT command: files that are currently open are copied to disc (ASCII files), work files are saved (".e" files), and the editing session is terminated.

FILL LINE

This operation attempts to place as many words as possible on a 70 character line. For short lines, words are taken from the next text line. To truncate a too lengthy line, words are moved to a new line (which becomes the current line).

FIRST PAGE

This operation moves the window back to the first page in the text file. A page represents the number of lines that constitute a "screenful." The first text line becomes the current line.

HOME

This operation positions the cursor in the upper left-hand corner of the text entry area (column 1, row 1) of the current window.

INSERT BLOCK

This operation effectively pushes the current line to the bottom of the window by inserting a block of blank lines before it.

LAST PAGE

This operation moves the window forward to the last page in the text file. The last text line becomes the current line.

LINE DELETE

This operation deletes the current line. (RESTORE restores a line deleted by this operation.)

LINE ERASE

This operation replaces the characters from the cursor position to the end of the line with blanks.

LINE INSERT

This operation inserts a blank line before the current line. The blank line becomes the new current line.

MOVE BLOCK

This operation inserts a copy of the currently selected block before the current line, deletes the lines in the currently selected block, and then nullifies the current selection.

NEXT PAGE

This operation moves the window forward to the next page in the text file. The new current line occupies the same relative position in the window as did the previous current line.

PREVIOUS PAGE

This operation moves the window back to the preceding page in the text file. The new current line occupies the same relative position in the window as did the old current line.

RESTORE (UNDO)

This operation restores the last line deleted by the LINE DELETE key by inserting the line just before the current line. The restored line becomes the new current line.

RETURN

This operation moves the cursor to the beginning of the next line. If the next line is not displayed, the window is moved forward by one line.

SELECT

This operation and the SELECT command perform identically. See the Editor Command section for a full description; briefly, SELECT identifies the current line by line number, and defines top and bottom lines for a block of text.

TAB

This operation moves the cursor to the next predefined tab position. If TAB is executed past the last identified tab, the cursor is moved to column 1 of the next line. If the next line is not displayed, the window is moved forward in the text file one line.

WINDOW A
WINDOW B
WINDOW C

These operations select the specified window as the current window. They are identical to the WINDOW editing commands.

CHAPTER 3
REDIT TUTORIAL

INTRODUCTION

This tutorial takes the user through the basics of creating and editing a text file. No attempt is made to illustrate all the features of the text editor, simply some basics. While the Monochrome Display and the Text Terminal support the text editor in the same way, display format differences were shown in Chapter 1; and keyboard and functional differences were noted in Chapter 3. In this tutorial a few further differences are encountered and explained.

INVOKING THE EDITOR

To invoke the text editor on either the Monochrome Display or Text Terminal, type after the operating system prompt "$":

$ redit

The screen now displays in the first line (the advisory and command area line) the following:

Ridge Text Editor (07-Sep-83):

The date indicates the version of the editor. The cursor appears after the colon in the command input area.

ATTACHING A FILE

ATTACH brings an existing file to the screen--or it can be used to create a new file. To edit an existing file ("stengel"), the command line should look like this:

Ridge Text Editor (07-Sep-83): ATTACH stengel

As is true for all commands entered into the command status area, you must press the ENTER/EXECUTE key when you want the command to be executed.

The file now appears on the screen, in window A by default. Window A occupies the entire screen unless windows B and/or C are attached; Figure 4 in Chapter 2 illustrates how the screen is subdivided.

To position the file in one of the other windows, specify ",b" or ",c" after the file name:

Ridge Text Editor (07-Sep-83): ATTACH stengel,b

To create a new file (named "gold"), ",new" must be added after the file name:

Ridge Text Editor (07-Sep-83): ATTACH gold,new

However, the quickest way to start the editor and get going on a file is to use a file name when invoking the editor. This causes ATTACH to be executed automatically. To create a new file, type:

```
$ redit gold,new
```

Windows can, of course, be specified. To call "stengel" into window C, you would type:

```
$ redit stengel,c
```

After the ATTACH command has been executed, a banner appears in the window header and selection status line (the second screen line) of the display. For the new file "gold", this line shows the file name, the currency indicator ("@") since the window is the current one, and the current line number:

```
Ridge Text Editor (07-Sep-83):ATTACH gold,new
gold@1
```

Text can now be entered in typewriter fashion. Assume the following three line file is entered:

```
Ridge Text Editor (07-Sep-83):ATTACH gold,new
gold@1
The words of Samual Goldwyn himself, best illustrate the
goldwynism, a type of mixed metaphor: "No oral contract is
worth the paper it's written on."
```

As a strike-over editor, corrections can be made by simply writing over existing text. On the Monochrome Display, pressing the REPT key simultaneously with a cursor key speeds up cursor movement.

BLOCK MANIPULATION

Assume now that the existing "stengel" file should be appended to "gold". Use the ENTER/EXIT key to enter the command input area and then type:

```
Ridge Text Editor (07-Sep-83):attach stengel,b
```

After pressing the ENTER/EXIT key again, the following appears in window B:

(window A)

```
stengel@1
Named for baseball manager Casey Stengel, a stengelism is
yet another marvelously confused metaphor, to wit: "He's so
lucky he'd fall in a hole and come up with a silver spoon."
```

(window B)

The window's status line gives the file name, currency status indicator (B is now the current window), and the current line number ("1"). Editing is performed in the B and C windows in the same way as it is in window A. So the file could be added to, deleted from, etc.

The easiest way to copy the file to "gold" is to use the COPY command. First, the line numbers of the text to be copied must be determined. Since window B is the current window, position the cursor on the first line of the file to be copied (line 1), and press SELECT. The window A status lines looks like this:

```
Ridge Text Editor (07-Sep-83):attach stengel,b
goldstengel1..1
```

Note that the currency indicator is absent from window A and that no top line is indicated for that window. "stengel" is shown to have a current block defined in it (a block can be one line). The "1.." represents the top line of the block, and "1" the bottom line of the block. Move the cursor to the last text line to be copied (line 3). Press SELECT. The window A status line now appears as:

```
Ridge Text Editor (07-Sep-83):attach stengel,b
gold            stengel1..3
```

The bottom line is now seen to be 3. Next, return to window A by pressing the WINDOW A key, and locating the cursor on the line below which the copied text should appear (here, it will be line 4). Next, simply press the COPY key. The following now appears in window A:

```
Ridge Text Editor (07-Sep-83):
gold@1
The words of Samual Goldwyn himself, best illustrate the
goldwynism, a type of mixed metaphor: "No oral contract is
worth the paper it's written on."
Named for baseball manager Casey Stengel, a stengelism is
yet another marvelously confused metaphor, to wit: "He's so
lucky he'd fall in a hole and come up with a silver spoon."
```

If the "stengel" portion of the text should appear before the "gold" portion of the file, the MOVE <range> command can be used in just the same way as the COPY <range> was used. The difference between the two is that MOVE deletes the original lines. MOVE BLOCK could also be used to reorganize the text. It is similar to COPY BLOCK except that it deletes the original lines.

If this were a longer file and the word "stengelism" were misspelled throughout, the REPEAT command could be used to perform a global search and replace. Enter the command input area and type:

```
repeat (find "stengalism"; replace "stengelism") count -1
```

Also, ", quiet" can be added onto the command sequence so that the text manipulation is done without updating the screen after each change.

LEAVING THE EDITOR

A file can be closed and the editor terminated by using the EXIT key. All open files are saved, updated, and control is returned to the operating system.

COMMENTS ON ASCII AND WORK FILES

If an ASCII file is lost (for example, moved to another directory), the file may still be edited in the current directory as long as the work file exists. Do not invoke the file by its ".e" name, however, since the work file is not readable by the editor; rather, use the ASCII file name ("prime.s"); make some sort of change to the file; then save the file. Now, both an ASCII and a work file should appear in the directory.

A work file appears not to be updated if it is manipulated by non-redit commands (e.g., with CAT or vi). For instance, assume the "stengel" file, initially created using redit, is modified using vi. If the file "red" is concatenated out onto the screen, changes made using vi appear. However, if "stengel" is invoked with redit, the original version appears. Therefore, to call up the most recent version, first remove the file "stengel.e" from the directory.

ED – a text editor

Introduction

Ed is a "text editor", an interactive program for creating and modifying text, that accepts directions from a user at a terminal. The text is often a document like this one, or a program or perhaps data for a program.

This introduction simplifies learning *ed.* Read this document and use *ed* to follow the examples. (It is useful to solicit advice from experienced users.)

Do the exercises! They cover material not completely discussed in the actual text. An appendix summarizes the commands.

This is an introduction and a tutorial, and only the most useful and most frequently used parts of *ed* are covered here. After you master this tutorial, try *Advanced Ed.* This tutorial assumes you know the basic ROS procedures, like logging on and using the file system.

Getting Started

Log in to your system and wait for the $ prompt. The easiest way to get *ed* is to type:

```
ed        (followed by a return)
```

You are now ready to go – *ed* is waiting for you to tell it what to do.

Terminology: In *ed* jargon, the text being worked on is said to be ''kept in a buffer.'' Think of the buffer as a work space, the piece of paper on which we will write and modify things. Terminology: The user tells *ed* what to do to his text by typing instructions called ''commands.'' Most commands consist of a single letter, which must be typed in lower case. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected – we will discuss these shortly.) *Ed* makes no response to most commands – there is no prompting or typing of messages like ''ready''. (This silence is preferred by experienced users, but is sometimes a hangup for beginners.)

Creating Text – the Append command "a"

Suppose you want to create some text starting from scratch. Perhaps you are typing the very first draft of a paper (that will undergo modifications later.) This shows how to get some text in. Later, we'll talk about how to change it.

When *ed* is first started, it is like working with a blank piece of paper – there is no text or information present. This must be supplied by the person using *ed;* it is usually done by typing the text.

The first command is *append,* written as the letter

```
a
```

all by itself. It means ''append (or add) text lines to the buffer, as I type them in.'' Appending is like writing fresh material on a piece of paper.

To enter lines of text into the buffer, just type an **a** followed by a RETURN, followed by the lines of text you want, like this:

```
a
Now is the time
for all good men
to come to the aid of their party.
```

The only way to stop appending is to type a line that contains only a period. The ''.'' tells *ed* that you have finished appending. (Even experienced users forget that terminating ''.'' sometimes. If *ed* seems to be ignoring you, type an extra line with just ''.'' on it. You may then find you've added some garbage lines to your text, which you'll have to take out later.)

After the append command has been done, the buffer will contain the three lines

```
Now is the time
for all good men
to come to the aid of their party.
```

The "**a**" and "." aren't there, because they are not text.

To add more text to what you already have, just issue another **a** command, and continue typing.

Error Messages – "?"

If at any time you make an error in the commands you type to *ed*, it will tell you by displaying

```
?
```

This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed.

Writing text out as a file – the Write command "w"

It's likely that you'll want to save your text for later use. To write the contents of the buffer onto a file, use the *write* command

```
w
```

followed by the filename under which you want to store the text. This copies the buffer's contents onto the specified file (destroying any previous information in the file). To save the text on a file named **junk**, for example, type

```
w junk
```

Leave a space between **w** and the file name. *Ed* will respond by printing the number of characters it wrote out. In this case, *ed* would respond with

```
68
```

(Remember that blanks and the return character at the end of each line are included in the character count.)

Writing a file makes a copy of the text, but the buffer's contents are not disturbed, so you can continue adding lines to it. This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a **w** command. (Writing out the text onto a file from time to time as it is being (Writing the text into a file from time to time as it is being created is a good idea. If the system crashes or if you make a horrible mistake, you will lose all the text in the buffer, but any text in a file is safe.)

Leaving ed – the Quit command "q"

To terminate a session with *ed*, save the text you're working on by writing it into a file using the **w** command, and then type the command

```
q
```

which stands for *quit*. When you quit, the buffer and all its contents vanish. In this rare case, the computer protects you from quitting before you have written the file; you may not quit if the editor detects that the buffer has been modified since the last use of "w". To override the protection, type "q!" to quit and lose your buffer.

Exercise 1:

Enter *ed* and create some text using

```
a
. . . text . . .
.
```

Write it out using **w**. Then leave *ed* with the **q** command, and print the file, to see that everything worked. (To print a file, type

```
pr filename
```

or

```
cat filename
```

in response to the $ prompt. Try both.)

Reading text from a file – the Edit command "e"

Another common way to get text into the buffer is to read it from a file that already exists. This is what you do to edit text that you saved with the **w** command in a previous session. The *edit* command **e** fetches the entire contents of a file into the buffer. If you had saved the three-line "Now is the time", file, the *ed* command

```
e junk
```

would fetch the entire contents of the file **junk** into the buffer, and respond

```
68
```

which is the number of characters in **junk**. *If anything was already in the buffer, it is deleted first.*

If you use the **e** command to read a file into the buffer, you need not use a file name after a subsequent **w** command; *ed* remembers the last file name used in an **e** command, and **w** will write to the same file.
A good way to operate is

```
ed
e file
[editing session]
w
q
```

This way, you can simply type **w** from time to time, and be secure in the knowledge that if you got the file name right at the beginning, you are writing into the proper file each time.

To find out at any time what file name *ed* is remembering, type the *file* command **f**. In this example, type

```
f
```

and replies

```
junk
```

Reading text from a file – the Read command "r"

Sometimes you want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command **r**.

```
r junk
```

reads the file **junk** into the buffer; it adds it to the end of whatever is already in the buffer. If you do a read after an edit:

```
e junk
r junk
```

the buffer will contain *two* copies of the text (six lines).

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

Like the **w** and **e** commands, **r** prints the number of characters read in, after the reading operation is complete.

Generally, **r** is used less than **e**.

Exercise 2:

Experiment with the **e** command – try reading and printing various files. You may get an error **?name**, where **name** is the name of a file; this means that the file doesn't exist, usually because you spelled the file name wrong, or because you are not allowed to read or write it. Verify that reading and appending work similarly. Verify that

```
ed filename
```

is exactly equivalent to

```
ed
e filename
```

What does

```
f filename
```

do?

Printing the contents of the buffer – the Print command "p"

To *print* or list the contents of the buffer (or parts of it) on the terminal, use the print command

```
p
```

The way this is done is as follows. Specify the lines where you want printing to begin and where you want it to end, separated by a comma, and followed by the letter **p**. Thus, to print the first two lines of the buffer, (lines 1 through 2) type

```
1,2p
```

Ed will respond with

```
Now is the time
for all good men
```

If you want to print *all* the lines in the buffer, type **1,3p** as above if you know there are exactly 3 lines in the buffer. But in general, you don't know how many there are, so what do you use for the ending line number? *Ed* provides a shorthand symbol for "line number of last line in buffer" – the dollar sign $. Use it this way:

```
1,$p
```

This will print *all* the lines in the buffer (line 1 to last line.) If you want to stop the printing before it is finished, push the DEL or Delete key; *ed* displays

```
?
```

and waits for the next command.

To print the *last* line of the buffer, you could type

```
$,$p
```

but *ed* lets you abbreviate this to

```
$p
```

You can print any single line by typing the line number followed by a **p**. Thus

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

In fact, *ed* lets you abbreviate even further: you can print any single line by typing *just* the line number – no need to type the letter **p**. If you type

```
$
```

ed will print the last line of the buffer.

You can also use $ in combinations like

```
$– 1,$p
```

which prints the last two lines of the buffer. This helps when you want to see how far you got in typing.

Exercise 3:

As before, create some text using the **a** command and experiment with the **p** command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that you cannot print a buffer in reverse order by entering "3.1p".

The current line – "Dot" or "."

Suppose your buffer still contains the six lines as above, that you have just typed

```
1,3p
```

and *ed* has printed the three lines for you. Try typing just

```
p
```

This will display

```
to come to the aid of their party.
```

which is the third line of the buffer. In fact, it is the last (most recent) line that you have done anything with. (You just printed it!) You can repeat this **p** command without line numbers, and it will continue to print line 3.

The reason is that *ed* maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

```
.          (pronounced "dot").
```

Dot is a line number in the same way that $ is; it means exactly "the current line", or "the line you most recently did something to." You can use it in several ways – one possibility is to enter

```
.,$p
```

This will print all the lines from (including) the current line to the end of the buffer. In our example, these are lines 3 through 6.

Some commands change the value of dot, while others do not. The **p** command sets dot to the number of the last line printed; the last command will set both . and $ to 6.

Dot is most useful when used in combinations like

```
. .+ 2 or .+ 1p
```

This means "print the next line" and is a handy way to step slowly through a buffer. You can also enter

```
.– 1     (or .– 1p )
```

which means "print the line *before* the current line." This enables you to go backwards.

```
.– 3,.– 1p
```

prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing

```
.=
```

Ed will respond by printing the value of dot.

Let's summarize some things about the **p** command and dot. **p** can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given (with or without the letter **p**), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second.

Typing a single return will cause printing of the next line. It's equivalent to **.+1p**. Try it. Try typing a you will find that it's equivalent to **.– 1p**.

Deleting lines: the "d" command

To get rid of the three extra lines in the buffer, use the *delete* command

```
d
```

Except that **d** deletes lines instead of printing them, its action is similar to that of **p**. The lines to be deleted are specified for **d** exactly as they are for **p**:

starting line, ending line d

Thus

```
4,$d
```

deletes lines 4 through the end. There are now three lines left, as you can check by using

```
1,$p
```

And notice that $ now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to $.

Exercise 4:

Experiment with **a**, **e**, **r**, **w**, **p** and **d** until you are sure that you know what they do, and until you understand how dot, $, and line numbers are used.

If you are adventurous, try using line numbers with **a**, **r** and **w** as well. You will find that **a** will append lines *after* the line number that you specify (rather than after dot); that **r** reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and that **w** will write out exactly the lines you specify, not necessarily the whole buffer.

These variations are handy. You can insert a file at the beginning of a buffer by entering

```
0r filename
```

and you can enter lines at the beginning of the buffer by typing

```
0a
. . . text . . .
.
```

Notice that **.w** is *very* different from

```
.
w
```

Modifying text: the Substitute command "s"

```
s
```

This important command is used to change individual words or letters within a line or group of lines. It is what you use, for example, for correcting spelling mistakes and typing errors.

Suppose that by a typing error, line 1 says

```
Now is th time
```

– The *e* has been left off *the*. You can use **s** to fix this up as follows:

```
1s/th/the/
```

This says: "in line 1, substitute for the characters *th* the characters *the*." To verify that it works (*ed* will not print the result automatically) enter

```
p
```

and get

```
Now is the time
```

which is what you wanted. Notice that dot must have been set to the line where the substitution took place, since the **p** command printed that line. Dot is always set this way with the **s** command.

The general way to use the substitute command is

starting-line, ending-line s/*change this*/*to this*/

in *all* the lines between *starting-line* and *ending-line,* the string of characters between the first pair of slashes is replaced by whatever is between the second pair, Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for **p**, except that dot is set to the last line changed. (But if no substitution took place, dot is *not* changed. This causes an error **?** as a warning.)

You can enter

```
1,$s/speling/spelling/
```

to correct the first spelling mistake on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the **s** command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

```
s/something/something else/p
```

which makes some correction on the current line, and then prints it, to make sure it worked out right. If it didn't, you can try again. (Notice that there is a **p** on the same line as the **s** command. With few exceptions, **p** can follow any command; no other multi-command lines are legal.)

It's also legal to enter

```
s/something//
```

which means "change the first string of characters to "*nothing*", i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. If you have

```
Nowxx is the time
```

you can enter

```
s/xx//p
```

to get

```
Now is the time
```

Notice that **//** (two adjacent slashes) means "no characters", not a blank. There *is* a difference! (See below for another meaning of **//**.)

Exercise 5:

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this:

```
a
the other side of the coin
.
s/the/on the/p
```

You will get

```
on the other side of the coin
```

A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a **g** (for "global") to the **s** command, like this:

```
s/ something/differen/gp
```

Try other characters instead of slashes to delimit the two sets of characters in the **s** command – anything should work except blanks or tabs.

You will get funny results using any of the characters

```
^   .   $   [   *   \   &
```

In the first or second string. read the section on ''Special Characters''.

Context searching – "/ . . . /"

With the substitute command mastered, you can move on to another highly important idea of *ed* – context searching.

Suppose you have the original three line text in the buffer:

```
Now is the time
for all good men
to come to the aid of their party.
```

Suppose you want to find the line that contains *their* so you can change it to *the*. Now with only three lines in the buffer, it's pretty easy to keep track of what line the word *their* is on. But if the buffer contained several hundred lines, you would no longer know what this line number would be. Context searching is a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

The way to say ''search for a line that contains this particular string of characters'' is to type

/string of characters we want to find/

For example, the *ed* command

```
/their/
```

is a context search which is sufficient to find the desired line. – It locate the next occurrence of the characters between slashes (''their''). It also sets dot to that line and prints the line for verification:

```
to come to the aid of their party.
```

''Next occurrence'' means that *ed* starts looking for the string at line **.+1**, searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search ''wraps around'' from $ to 1.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can't be found in any line, *ed* types the error message

```
?
```

Otherwise it prints the line it found.

You can do both the search for the desired line *and* a substitution all at once

```
/their/s/their/the/p
```

which will yields

```
to come to the aid of the party.
```

There were three parts to that last command: context search for the desired line, make the substitution, print the line.

The expression **/their/** is a context search expression. In its simplest form, a context search expression is a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like **s**. They were used both ways in the examples above.

Suppose the buffer contains the three familiar lines

```
Now is the time
for all good men
to come to the aid of their party.
```

Then the *ed* line numbers

```
/Now/+1
/good/
/party/–1
```

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, you could say

```
/Now/+ 1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/- 1s/good/bad/
```

The choice is dictated only by convenience. You could print all three lines by

```
/Now/,/party/p
```

or

```
/Now/,/Now/+ 2p
```

The basic rule is: a context search expression is *the same as* a line number, so it can be used wherever a line number is needed.

Exercise 6:

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print, and delete commands. (They can also be used with **r**, **w**, and **a**.)

Try context searching using **?text?** instead of **/text/**. This scans lines in the buffer in reverse order rather than normal. This is sometimes useful if you go too far while looking for a string of characters – it's an easy way to back up.

(If you get funny results with any of the characters

```
^   .   $   [   *   \   &
```

read the section on "Special Characters".)

Ed provides a shorthand for repeating a context search for the same string. For example, the *ed* line number

```
/string/
```

will find the next occurrence of **string**. It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely

```
//
```

This shorthand stands for "the most recently used context search expression." It can also be used as the first string of the substitute command, as in

```
/string1/s//string2/
```

which will find the next occurrence of **string1** and replace it by **string2**. This can save a lot of typing. Similarly

```
??
```

means "scan backwards for the same expression."

Change and Insert – "c" and "i"

This section discusses the *change* command

```
c
```

which is used to change or replace a group of one or more lines, and the *insert* command

```
i
```

which is used for inserting a group of one or more lines.

"Change", written as

```
c
```

replaces a number of lines with different text, which are typed in at the terminal. For example, to change lines **.+1** through $ to something else, type

```
.+1,$c
. . . type the lines of text you want here . . .
.
```

The lines you type between the **c** command and the **.** will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors in them.

If only one line is specified in the **c** command, just that line is replaced. (You can type in as many replacement lines as you like.) Notice that **.** ends the input this works just like the **.** in the append command and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

"Insert" is similar to append.

```
/string/i
. . . type the lines to be inserted here . . .
.
```

will insert the given text *before* the next line that contains string. The text between **i** and **.** is *inserted before* the specified line. If no line number is specified, dot is used. Dot is set to the last line inserted.

Exercise 7:

"Change" is like a combination of delete followed by insert. Experiment to verify that

```
start, end d
i
. . . text . . .
.
```

is almost the same as

```
start, end c
. . . text . . .
.
```

These are not *precisely* the same if line $ gets deleted. Check this out. What is dot?

Experiment with **a** and **i**, to see that they are similar, but not the same. You will observe that

```
line-number a
. . . text . . .
.
```

appends *after* the given line, while

```
line-number i
. . . text . . .
.
```

inserts *before* it. Observe that if no line number is given, **i** inserts before line dot, while **a** appends after line dot.

Moving text around: the "m" command

The move command **m** lets you move a group of lines from one place to another. If you want to put the first three lines of the buffer at the end, enter

```
1,3w temp
$r temp
1,3d
```

Or a lot easier with the **m** command

```
1,3m$
```

The general case is

start- line, end- line m *after- this- line*

Of course, the lines to be moved can be specified by context searches; if you have

```
First paragraph
. . .
end of first paragraph.
Second paragraph
. . .
end of second paragraph.
```

you can reverse the two paragraphs by

```
/Second/,/end of second/m/First/- 1
```

Notice the – **1.** the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

The global commands "g" and "v"

The *global* command **g** executes one or more *ed* commands on all those lines in the buffer that match some specified string. For example,

```
g/peling/p
```

prints all lines that contain **peling**.

```
g/peling/s//pelling/gp
```

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

```
1,$s/peling/pelling/gp
```

which only prints the last line substituted. Another subtle difference is that the **g** command does not give a **?** if **peling** is not found, but the **s** command does.

There may be several commands (including **a**, **c**, **i**, **r**, **w**, but not **g**); in that case, every line except the last must end with a backslash \

```
g/xxx/.- 1s/abc/def/B
.+ 2s/ghi/jkl/B
.- 2,.p
```

makes changes in the lines before and after each line that contains **xxx**, then prints all three lines.

The **v** command is the same as **g**, except that the commands are executed on every line that does *not* match the string following **v**:

```
v/ /d
```

deletes every line that does not contain a blank.

Special Characters

You may have noticed that things just don't work right when you used some characters like ., *, $, and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only,* . means "any character," not a period, so

```
/x.y/
```

means "a line with an **x**, *any character,* and a **y**," *not* just "a line with an **x**, a period, and a **y**." A complete list of the special characters that can cause trouble is the following:

```
^   .   $   [   *   \
```

Warning: The backslash character \ is special to *ed*. For safety's sake, avoid it. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

```
s/\\\.\*/backslash dot star/
```

will change \.* into "backslash dot star".

Here is a hurried synopsis of the other special characters. First, the circumflex ^ signifies the beginning of a line. Thus

```
/^string/
```

finds **string** only if it is at the beginning of a line: it will find

```
string
```

but not

```
the string...
```

The dollar-sign $ is just the opposite of the circumflex; it means the end of a line:

```
/string$/
```

will only find an occurrence of **string** that is at the end of some line. This implies, of course, that

```
/^string$/
```

will find only a line that contains just **string**, and

```
/^.$/
```

finds a line containing exactly one character.

The character **.**, as we mentioned above, matches anything

```
/x.y/
```

matches any of

```
x+y
x-y
x y
x.y
```

This is useful in conjunction with *, which is a repetition character; **a*** is a shorthand for "any number of **a**'s," so **.*** matches any number of anythings. This is used like this:

```
s/.*/stuff/
```

which changes an entire line, or

```
s/.*,//
```

which deletes all characters in the line up to and including the last comma. (Since **.*** finds the longest possible match, this goes up to the last comma.)

[is used with] to form "character classes"; for example,

```
/[0123456789]/
```

matches any single digit – any one of the characters inside the braces will cause a match. This can be abbreviated to **[0–9]**.

Finally, the & is another shorthand character – it is used only on the second part of a substitute command where it means "whatever was matched on the first part". It saves typing. Suppose the current line is

```
Now is the time
```

and you wantto put parentheses around it. You could retype the line, but this is tedious, Or enter

```
s/^/(/
s/$/)/
```

using your knowledge of ^ and $. But the easiest way uses the &:

```
s/.*/(&)/
```

This says "match the whole line, and replace it by itself surrounded by parentheses." The & can be used several times in a line; consider using

```
s/.*/&? &!!/
```

to produce

Now is the time? Now is the time!!

You don't have to match the whole line, of course. if the buffer contains

the end of the world

you can type

```
/world/s//& is at hand/
```

to produce

the end of the world is at hand

Observe this expression carefully, because it illustrates how to take advantage of *ed* to save typing. The string **/world/** found the desired line; the shorthand // found the same word in the line; and the & saves you from typing it again.

The & is a special character only within the replacement text of a substitute command, and has no special meaning elsewhere. You can turn off the special meaning of & by preceding it with a \:

```
s/ampersand/\&/
```

will convert the word "ampersand" into the literal symbol & in the current line.

Summary of Commands and Line Numbers

The general form of *ed* commands is the command name, perhaps preceded by one or two line numbers, and, in the case of **e**, **r**, and **w**, followed by a file name. Only one command is allowed per line, but a **p** command may follow any other command (except for **e**, **r**, **w** and **q**).

a: Append, add lines to the buffer (at line dot, unless a different line is specified). Appending continues until . is typed on a new line. Dot is set to the last line appended.

c: Change the specified lines to the new text which follows. The new lines are terminated by a ., as with **a**. If no lines are specified, replace line dot. Dot is set to last line changed.

d: Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless $ is deleted, in which case dot is set to $.

e: Edit new file. Any previous contents of the buffer are thrown away, so issue a **w** beforehand.

f: Print remembered filename. If a name follows **f** the remembered name will be set to it.

g: The command

```
g/---/commands
```

will execute the commands on those lines that contain **---**, which can be any context search expression.

i: Insert lines before specified line (or dot) until a . is typed on a new line. Dot is set to last line inserted.

m: Move lines specified to after the line named after **m**. Dot is set to the last line moved.

p: Print specified lines. If none specified, print line dot. A single line number is equivalent to *line-number* **p**. A single return prints **.+1**, the next line.

q: Quit *ed*. Erases all text in buffer if you enter it twice without first giving a **w** command.

r: Read a file into buffer (at end unless specified elsewhere.) Dot set to last line read.

s: The command

```
s/string1/string2/
```

substitutes the characters **string1** into **string2** in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. **s** changes only the first occurrence of **string1** on a line; to change all of them, type a **g** after the final slash.

v: The command

```
v/---/commands
```

executes **commands** on those lines that *do not* contain **---**.

w: Write out buffer onto a file. Dot is not changed.

.=: Print value of dot. (= by itself prints the value of $.)

!: The line

```
!command-line
```

causes **command-line** to be executed as a ROS command.

/-----/: Context search. Search for next line which contains this string of characters. Print it. Dot is set to the line where string was found. Search starts at **.+1**, wraps around from $ to 1, and continues to dot, if necessary.

?-----?: Context search in reverse direction. Start search at **.– 1**, scan to 1, wrap around to $.

Advanced Ed

This document is based on a paper by Brian W. Kernighan of Bell Laboratories.

1. INTRODUCTION

This is a sequel to *Ed – a Text Editor.* This assumes the reader is familiar with ROS basics, and covers topics like special characters in search and substitute commands, line addressing, global commands, and line moving and copying. Effective use of related file manipulation tools, like grep(1) and sed(1), is discussed briefly.

This document gives you ideas of new editing techniques to try, but you will not learn them unless you do.

2. SPECIAL CHARACTERS

The next few sections discuss shortcuts and labor-saving devices. Remember, try them and your confidence in using them will increase.

The List command 'l'

ed provides two commands for printing the contents of the lines you're editing. Most people are familiar with **p**, in combinations like

```
1,$p
```

to print all the lines you're editing, or

```
s/abc/def/p
```

to change 'abc' to 'def' on the current line. Less familiar is the *list* command **l** (the letter '*l*'), which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, **l** will print each tab as ≫ and each backspace as ≺. This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The **l** command also 'folds' long lines for printing — any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash \, so you can tell it was folded. This is useful for printing long lines on short terminals.

Occasionally the **l** command will print in a line a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don't print, like form feed or vertical tab or bell. Each such combination is a single character. When you see such characters, be wary — they may have surprising meanings when printed on some terminals. Often their presence means that your finger slipped while you were typing; you almost never want them.

The Substitute Command 's'

Most of the next few sections will be taken up with a discussion of the substitute command **s**. Since this is the command for changing the contents of individual lines, it probably has the most complexity of any **ed** command, and the most potential for effective use.

As the simplest place to begin, recall the meaning of a trailing **g** after a substitute command. With

```
s/this/that/
```

and

```
s/this/that/g
```

the first one replaces the *first* 'this' on the line with 'that'. If there is more than one 'this' on the line, the second form with the trailing **g** changes *all* of them.

Either form of the **s** command can be followed by **p** or **l** to 'print' or 'list' (as described in the previous section) the contents of the line:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

are all legal, and mean slightly different things. Make sure you know what the differences are.

Of course, any **s** command can be preceded by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus

```
1,$s/mispell/misspell/
```

changes the *first* occurrence of 'mispell' to 'misspell' on every line of the file. But

```
1,$s/mispell/misspell/g
```

changes *every* occurrence in every line (and this is more likely to be what you wanted in this particular case).

You should also notice that if you add a **p** or **l** to the end of any of these substitute commands, only the last line that got changed will be printed, not all the lines. We will talk later about how to print all the lines that were modified.

The Undo Command 'u'

Occasionally you will make a substitution in a line, only to realize too late that it was a ghastly mistake. The 'undo' command **u** lets you 'undo' the last substitution: the last line that was substituted can be restored to its previous state by typing the command

```
u
```

The Metacharacter '.'

As you have undoubtedly noticed when you use **ed**, certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. In the next several sections, we will talk about these special characters, which are often called 'metacharacters'.

The first one is the period '.'. On the left side of a substitute command, or in a search with '/.../', '.' stands for *any* single character. Thus the search

```
/x.y/
```

finds any line where 'x' and 'y' occur separated by a single character, as in

```
x+y
x-y
x y
x.y
```

and so on. (We will use   to stand for a space whenever we need to make it visible.)

Since '.' matches a single character, that gives you a way to deal with funny characters printed by **l**. Suppose you have a line that, when printed with the **l** command, appears as

```
.... th\07is ....
```

and you want to get rid of the \07 (which represents the bell character, by the way).

The most obvious solution is to try

```
s/\07//
```

but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '.' comes in handy. Since '\07' really represents a single character, if we say

```
s/th.is/this/
```

the job is done. The '.' matches the mysterious character between the 'h' and the 'i', *whatever it is.*

Bear in mind that since '.' matches any single character, the command

```
s/./,/
```

converts the first character on a line into a ',', which very often is not what you intended.

As is true of many characters in **ed**, the '.' has several meanings, depending on its context. This line shows all three:

```
.s/./.
```

The first '.' is a line number, the number of the line we are editing, which is called 'line dot'. (We will discuss line dot more in Section 3.) The second '.' is a metacharacter that matches any single character on that line. The third '.' is the only one that really is an honest literal period. On the *right* side of a substitution, '.' is not special. If you apply this command to the line

```
Now is the time.
```

the result will be

```
.ow is the time.
```

which is probably not what you intended.

The Backslash '\'

The period '.' means 'any character', so what if you really want a period? For example, how do you convert the line

```
Now is the time.
```

into

```
Now is the time?
```

The backslash '\' does the job. A backslash turns off any special meaning that the next character might have; in particular, '\.' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in

```
Now is the time.
```

like this:

```
s/\./?/
```

The pair of characters '\.' is considered by **ed** to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

```
.PP
```

The search

```
/.PP/
```

isn't adequate, for it will find a line like

```
THE APPLICATION OF ...
```

because the '.' matches the letter 'A'. But if you say

```
/\.PP/
```

you will find only lines that contain '.PP'.

The backslash can also be used to turn off special meanings for characters other than '.'. For example, consider finding a line that contains a backslash. The search

```
/\/
```

won't work, because the '\' isn't a literal '\', but instead means that the second '/' no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus

```
/\\/
```

does work. Similarly, you can search for a forward slash '/' with

```
/\//
```

The backslash turns off the meaning of the immediately following '/' so that it doesn't terminate the /.../ construction prematurely.

As an exercise, before reading further, find two substitute commands each of which will convert the line

```
\x\.\y
```

into the line

```
\x\y
```

Here are several solutions; verify that each works as advertised.

```
s/\\\.//
s/x../x/
s/..y/y/
```

A couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an **s** command: there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains a lot of slashes already, like

```
//exec //sys.fort.go // etc...
```

you could use a colon as the delimiter — to delete all the slashes, type

```
s:/::g
```

Second, if # and @ are your character erase and line kill characters, you have to type \# and \@ ; this is true whether you're talking to **ed** or any other program.

When you are adding text with **a** or **i** or **c**, backslash is not special, and you should only put in one backslash for each one you really want.

The Dollar Sign '$'

The next metacharacter, the '$', stands for 'the end of the line'. As its most obvious use, suppose you have the line

```
Now is the
```

and you wish to add the word 'time' to the end. Use the $ like this:

```
s/$/ time/
```

to get

```
Now is the time
```

Notice that a space is needed before 'time' in the substitute command, or you will get

```
Now is thetime
```

As another example, replace the second comma in the following line with a period without altering the first:

```
Now is the time, for all good men,
```

The command needed is

```
s/,$/./
```

The $ sign here provides context to make specific which comma we mean. Without it, of course, the **s** command would operate on the first comma to produce

```
Now is the time. for all good men,
```

As another example, to convert

```
Now is the time.
```

into

```
Now is the time?
```

as we did earlier, we can use

```
s/.$/?/
```

Like '.', the '$' has multiple meanings depending on context. In the line

```
$s/$/$/
```

the first '$' refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign, to be added to that line.

The Circumflex '^'

The circumflex (or hat or caret) '^' stands for the beginning of the line. For example, suppose you are looking for a line that begins with 'the'. If you simply say

```
/the/
```

you will in all likelihood find several lines that contain 'the' in the middle before arriving at the one you want. But with

```
/^the/
```

you narrow the context, and thus arrive at the desired one more easily.

The other use of '^' is of course to enable you to insert something at the beginning of a line:

```
s/^/ /
```

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains *only* the characters

```
.PP
```

you can use the command

```
/^\.PP$/
```

The Star '*'

Suppose you have a line that looks like this:

text x   y *text*

where *text* stands for lots of text, and there are some indeterminate number of spaces between the **x** and the **y**. Suppose the job is to replace all the spaces between **x** and **y** by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter '*' comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say

```
s/x *y/x y/
```

The construction ' *' means 'as many spaces as possible'. Thus 'x *y' means 'an x, as many spaces as possible, then a y'.

The star can be used with any character, not just space. If the original example was instead

text x– – – – – – – – – y *text*

then all '– ' signs can be replaced by a single space with the command

```
s/x– *y/x y/
```

Finally, suppose that the line was

text x..................y *text*

Can you see what trap lies in wait for the unwary? If you blindly type

```
s/x.*y/x y/
```

what will happen? The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management. Remember that '.' matches *any* single character? Then '.*' matches as many single characters as possible, and unless you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

text x *text* x................y *text* y *text*

then saying

```
s/x.*y/x y/
```

will take everything from the *first* 'x' to the *last* 'y', which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with '\.':

```
s/x\.*y/x y/
```

Now everything works, for '\.*' means 'as many *periods* as possible'.

There are times when the pattern '.*' is exactly what you want. For example, to change

```
Now is the time for all good men ....
```

into

```
Now is the time.
```

use '.*' to eat up everything after the 'for':

```
s/ for.*/./
```

There are a couple of additional pitfalls associated with '*' that you should be aware of. Most notable is the fact that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained

text xy *text* x   y *text*

and we said

```
s/x *y/x y/
```

the *first* 'xy' matches this pattern, for it consists of an 'x', zero spaces, and a 'y'. The result is that the substitute acts on the first 'xy', and does not touch the later one that actually contains some intervening spaces.

The way around this, if it matters, is to specify a pattern like

```
/x  *y/
```

which says 'an x, a space, then as many more spaces as possible, then a y', in other words, one or more spaces.

The other startling behavior of '*' is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The command

```
s/x*/y/g
```

when applied to the line

```
abcdef
```

produces

```
yaybycydyeyfy
```

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write

```
s/xx*/y/g
```

'xx*' is one or more x's.

The Brackets '[ ]'

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [and].

The construction

```
[0123456789]
```

matches any single digit — the whole thing is called a 'character class'. With a character class, the job is easy. The pattern '[0123456789]*' matches zero or more digits (an entire number), so

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and just to confuse the issue there are essentially no special characters inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can say

```
/[.\$^[]/
```

Within [...], the '[' is not special. To get a ']' into a character class, make it the first character.

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0−9]; similarly, [a−z] stands for the lower case letters, and [A−Z] for upper case.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. This is done by beginning the class with a '^':

```
[^0−9]
```

stands for 'any character *except* a digit'. Thus you might find the first line that doesn't begin with a tab or space by a search like

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that

```
/^[^^]/
```

finds a line that doesn't begin with a circumflex.

The Ampersand '&'

The ampersand '&' is used primarily to save typing. Suppose you have the line

```
Now is the time
```

and you want to make it

```
Now is the best time
```

Of course you can always say

```
s/the/the best/
```

but it seems silly to have to repeat the 'the'. The '&' is used to eliminate the repetition. On the *right* side of a substitute, the ampersand means 'whatever was just matched', so you can say

```
s/the/& best/
```

and the '&' will stand for 'the'. Of course this isn't much of a saving if the thing matched is just 'the', but if it is something truly long or awful, or if it is something like '.*' which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length,

```
s/.*/(&)/
```

The ampersand can occur more than once on the right side:

```
s/the/& best and & worst/
```

makes

```
Now is the best and the worst time
```

and

```
s/.*/&? &!!/
```

converts the original line into

```
Now is the time? Now is the time!!
```

To get a literal ampersand, naturally the backslash is used to turn off the special meaning:

```
s/ampersand/\&/
```

converts the word into the symbol. Notice that '&' is not special on the left side of a substitute, only on the *right* side.

Substituting Newlines

ed provides a facility for splitting a single line into two or more shorter lines by 'substituting in a newline'. As the simplest example, suppose a line has gotten unmanageably long because of editing (or merely because it was unwisely typed). If it looks like

text xy *text*

you can break it between the 'x' and the 'y' like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is typed on two lines. Bearing in mind that '\' turns off special meanings, it seems relatively intuitive that a '\' at the end of a line would make the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the **roff** or **nroff** formatting command '.ul'.

text a very big *text*

The command

```
s/ very /\
.ul\
very\
/
```

converts the line into four shorter lines, preceding the word 'very' by the line '.ul', and eliminating the spaces around the 'very', all at the same time.

When a newline is substituted in, dot is left pointing at the last line created.

Joining Lines

Lines may also be joined together, but this is done with the **j** command instead of **s**. Given the lines

```
Now is
  the time
```

and supposing that dot is set to the first of them, then the command

```
j
```

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

All by itself, a **j** command joins line dot to line dot+ 1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example,

```
1,$jp
```

joins all the lines into one big one and prints it. (More on line numbers in Section 3.)

Rearranging a Line with \(... \)

(This section should be skipped on first reading.) Recall that '&' is a shorthand that stands for whatever was matched by the left side of an **s** command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form

```
Smith, A. B.
Jones, C.
```

and so on, and you want the initials to precede the name, as in

```
A. B. Smith
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

The alternative is to 'tag' the pieces of the pattern (in this case, the last name, and the initials), and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \(and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol '\1' refers to whatever matched the first \(...\) pair, '\2' to the second \(...\), and so on.

The command

```
1,$s/^\([^,]*\), *\(.*\)/\2 \1/
```

although hard to read, does the job. The first \(...\) matches the last name, which is any string up to the comma; this is referred to on the right side with '\1'. The second \(...\) is whatever follows the comma and any spaces, and is referred to as '\2'.

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands **g** and **v** discussed in section 4 provide a way for you to print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

3. LINE ADDRESSING IN THE EDITOR

The next general area we will discuss is that of line addressing in **ed**, that is, how you specify what lines are to be affected by editing commands. We have already used constructions like

```
1,$s/x/y/
```

to specify a change on all lines. And most users are long since familiar with using a single newline (or return) to print the next line, and with

```
/thing/
```

to find a line that contains 'thing'. Less familiar, surprisingly enough, is the use of

```
?thing?
```

to scan *backwards* for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command.

Address Arithmetic

The next step is to combine the line numbers like '.', '$', '/.../' and '?...?' with '+' and '–'. Thus

```
$–1
```

is a command to print the next to last line of the current file (that is, one line before line '$'). For example, to recall how far you got in a previous editing session,

```
$–5,$p
```

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six, of course, you'll get an error message.

As another example,

```
.–3,.+3p
```

prints from three lines before where you are now (at line dot) to three lines after, thus giving you a bit of context. By the way, the '+' can be omitted:

```
.–3,.3p
```

is absolutely identical in meaning.

Another area in which you can save typing effort in specifying lines is to use '–' and '+' as line numbers by themselves.

```
–
```

by itself is a command to move back up one line in the file. In fact, you can string several minus signs together to move back up that many lines:

```
–––
```

moves up three lines, as does '–3'. Thus

```
–3,+3p
```

is also identical to the examples above.

Since '–' is shorter than '.–1', constructions like

```
–,.s/bad/good/
```

are useful. This changes 'bad' to 'good' on the previous line and on the current line.

'+' and '–' can be used in combination with searches using '/.../' and '?...?', and with '$'. The search

```
/thing/––
```

finds the line containing 'thing', and positions you two lines before it.

Repeated Searches

Suppose you ask for the search

```
/horrible thing/
```

and when the line is printed you discover that it isn't the horrible thing that you wanted, so it is necessary to repeat the search again. You don't have to re-type the search, for the construction

```
//
```

is a shorthand for 'the previous thing that was searched for', whatever it was. This can be repeated as many times as necessary. You can also go backwards:

```
??
```

searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use '//' as the left side of a substitute command, to mean 'the most recent pattern'.

```
/horrible thing/
 .... ed prints line with 'horrible thing' ...
s//good/p
```

To go backwards and change a line, say

```
??s//good/
```

Of course, you can still use the '&' on the right hand side of a substitute to stand for whatever got matched:

```
//s//&  &/p
```

finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then prints the line just to verify that it worked.

Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you issue a search command like

```
/thing/
```

you are left pointing at the next line that contains 'thing'. Then no address is required with commands like **s** to make a substitution on that line, or **p** to print it, or **l** to list it, or **d** to delete it, or **a** to append text after it, or **c** to change it, or **i** to insert text before it.

What happens if there was no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you were sitting on the only 'thing' when you issued the command. The same rules hold for searches that use '?...?'; the only difference is the direction in which you search.

The delete command **d** leaves dot pointing at the line that followed the last deleted line. When line '$' gets deleted, however, dot points at the *new* line '$'.

The line-changing commands **a**, **c** and **i** by default all affect the current line — if you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

a, **c**, and **i** behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say

```
a
 ... text ...
 ... botch ...          (minor error)
.
s/botch/correct/        (fix botched line)
a
 ... more text ...
```

without specifying any line number for the substitute command or for the second append command. Or you can say

```
a
 ... text ...
 ... horrible botch ...    (major error)
.
c                          (replace entire line)
 ... fixed up line ...
```

You should experiment to determine what happens if you add *no* lines with **a**, **c** or **i**.

The **r** command will read a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say **0r** to read a file in at the beginning of the text. (You can also say **0a** or **1i** to start adding text at the beginning.)

The **w** command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The **w** command does *not* change dot: the current line remains the same, regardless of what lines are written. This is true even if you say something like

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

Since the **w** command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you do something foolish, like clobbering what you're editing.

The least intuitive behavior, in a sense, is that of the **s** command. The rule is simple — you are left sitting on the last line that got changed. If there were no changes, then dot is unchanged.

To illustrate, suppose that there are three lines in the buffer, and you are sitting on the middle one:

```
x1
x2
x3
```

Then the command

```
– ,+ s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and print only the first line, and that is where dot would be set.

Semicolon ';'

Searches with '/.../' and '?...?' start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
```

Starting at line 1, one would expect that the command

```
/a/,/b/p
```

prints all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to print a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In **ed**, the semicolon ';' can be used just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in our example above, the command

```
/a/;/b/p
```

prints the range of lines from 'ab' to 'bc', because after the 'a' is found, dot is set to that line, and then 'b' is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the *second* occurrence of 'thing'. You could say

```
/thing/
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to say

```
/thing/;//
```

This says to find the first occurrence of 'thing', set dot to that line, then find the second and print only that.

Closely related is searching for the second previous occurrence of something, as in

```
?something?;??
```

Printing the third or fourth or ... in either direction is left as an exercise.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say

```
1;/thing/
```

because this fails if 'thing' occurs on line 1. But it is possible to say

```
0;/thing/
```

(one of the few places where 0 is a legal line number), for this starts the search at line 1.

Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you hit the interrupt or delete or rubout or break key while **ed** is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable — if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in some clean but unpredictable state in the middle (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus if you print until you see an interesting line, then hit delete, you are *not* sitting on that line or even near it. Dot is left where it was when the **p** command was started. ,

4. GLOBAL COMMANDS

The global commands **g** and **v** are used to perform one or more editing commands on all lines that either contain (**g**) or don't contain (**v**) a specified pattern.

As the simplest example, the command

```
g/UNIX/p
```

prints all lines that contain the word 'UNIX'. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

As another example, then,

```
g/^\./p
```

prints all the formatting commands in a file (lines that begin with '.').

The **v** command is identical to **g**, except that it operates on those line that do *not* contain an occurrence of the pattern. (Don't look too hard for mnemonic significance to the letter 'v'.) So

```
v/^\./p
```

prints all the lines that don't begin with '.' — the actual text lines.

The command that follows **g** or **v** can be anything:

```
g/^\./d
```

deletes all lines that begin with '.', and

```
g/^$/d
```

deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command, for this can be used to make a change and print each affected line for verification. For example, we could change the word 'Unix' to 'UNIX' everywhere, and verify that it really worked, with

```
g/Unix/s//UNIX/gp
```

Notice that we used '//' in the substitute command to mean 'the previous pattern', in this case, 'Unix'. The **p** command is done on every line that matches the pattern, not just those on which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** to use addresses, set dot, and so on, quite freely.

```
g/^\.PP/+
```

prints the line that follows each '.PP' command (the signal for a new paragraph in some formatting packages). Remember that '+' means 'one line past dot'. And

```
g/topic/?^\.SH?1
```

searches for each line that contains 'topic', scans backwards until it finds a line that begins '.SH' (a section heading) and prints the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally,

```
g/^\.EQ/+,/^\.EN/- p
```

prints all the lines that lie between lines beginning with '.EQ' and '.EN' formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

Multi-line Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then

```
g/thing/s/x/y/\
s/a/b/
```

is sufficient. The '\' signals the **g** command that the set of commands continues on the next line; it terminates on the first line that does not end with '\'. (As a minor blemish, you can't use a substitute command to insert a newline within a **g** command.)

You should watch out for this problem: the command

```
g/x/s//y/\
s/a/b/
```

does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be 'x' (as expected), and sometimes it will be 'a' (not expected). You must spell it out, like this:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute **a**, **c** and **i** commands under a global command; as with other multi-line constructions, all that is needed is to add a '\' at the end of each line except the last. Thus to add a '.nf' and '.sp' command before each '.EQ' line, type

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a '.' to terminate the **i** command, unless there are further commands being done under the global. On the other hand, it does no harm to put it in either.

5. CUT AND PASTE WITH ROS COMMANDS

Non-programmers often feel unconfident performing tasks like: changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

There operations are easy if you are careful. The next several sections talk about manipulating files with the file system. We will begin with the system commands for moving entire files around, then discuss **ed** commands for operating on pieces of files.

Changing the Name of a File

You have a file named 'memo' and you want it to be called 'paper' instead. How is it done?

The UNIX† command that renames files is called **mv** (for 'move'); it 'moves' the file from one name to another, like this:

```
mv  memo  paper
```

That's all there is to it: **mv** from the old name to the new name.

```
mv  oldname  newname
```

Warning: if there is already a file around with the new name, its present contents will be silently clobbered by the information from the other file. The one exception is that you can't move a file to itself —

```
mv  x  x
```

is illegal.

Making a Copy of a File

Sometimes you want a copy of a file — an entirely fresh version. This might be because you want to work on a file, and yet save a copy in case something gets fouled up, or just because you're paranoid.

Do it with the **cp** command, which stands for 'copy'. Suppose you have a file called 'good' and you want to save a copy before you make some dramatic editing changes. Choose a name — 'savegood' might be acceptable — then type

```
cp  good  savegood
```

This copies 'good' onto 'savegood', and you now have two identical copies of the file 'good'. (If 'savegood' previously contained something, it gets overwritten.)

Now if you decide at some time that you want to get back to the original state of 'good', you can say

```
mv  savegood  good
```

(if you're not interested in 'savegood' any more), or

```
cp  savegood  good
```

if you still want to retain a safe copy.

In summary, **mv** just renames a file; **cp** makes a duplicate copy. Both of them clobber the 'target' file if it already exists, so you had better be sure that's what you want to do *before* you do it.

†UNIX is a Trademark of Bell Laboratories.

Removing a File

If you decide you are really done with a file forever, you can remove it with the **rm** command:

```
rm  savegood
```

throws away (irrevocably) the file called 'savegood'.

Putting Two or More Files Together

The next step is the familiar one of collecting two or more files into one big one. This will be needed, for example, when the author of a paper decides that several sections need to be combined into one. The cleanest way to do it is with a command called **cat** which is short for 'concatenate', which is exactly what we want to do.

Suppose the job is to combine the files 'file1' and 'file2' into a single file called 'bigfile'. If you say

```
cat  file
```

the contents of 'file' will get printed on your terminal. If you say

```
cat  file1  file2
```

the contents of 'file1' and then the contents of 'file2' will *both* be printed on your terminal, in that order. So **cat** combines the files, all right, but it's not much help to print them on the terminal — we want them in 'bigfile'.

Fortunately, there is a way. You can tell the system that instead of printing on your terminal, you want the same information put in a file. The way to do it is to add to the command line the character > and the name of the file where you want the output to go. Then you can say

```
cat  file1  file2  >bigfile
```

and the job is done. (As with **cp** and **mv**, you're putting something into 'bigfile', and anything that was already there is destroyed.)

This ability to 'capture' the output of a program is one of the most useful aspects of the file system. Fortunately it's not limited to the **cat** program — you can use it with *any* program that prints on your terminal. We'll see some more uses for it in a moment.

Naturally, you can combine several files, not just two:

```
cat  file1  file2  file3  ...  >bigfile
```

collects a whole bunch.

Question: is there any difference between

```
cp  good  savegood
```

and

```
cat  good  >savegood
```

Answer: for most purposes, no. You might reasonably ask why there are two programs in that case, since **cat** is obviously all you need. The answer is that **cp** will do some other things as well, which you can investigate for yourself by reading the manual. For now we'll stick to simple usages.

Adding Something to the End of a File

Sometimes you want to add one file to the end of another. We have enough building blocks now that you can do it; in fact before reading further it would be valuable if you figured out how. To be specific, how would you use **cp**, **mv** and/or **cat** to add the file 'good1' to the end of the file 'good'?

You could try

```
cat good good1 >temp
mv temp good
```

which is probably most direct. You should also understand why

```
cat good good1 >good
```

doesn't work. (Don't practice with a good 'good'!)

The easy way is to use a variant of >, called >>. In fact, >> is identical to > except that instead of clobbering the old file, it simply tacks stuff on at the end. Thus you could say

```
cat good1 >>good
```

and 'good1' is added to the end of 'good'. (And if 'good' didn't exist, this makes a copy of 'good1' called 'good'.)

6. CUT AND PASTE WITH THE EDITOR

Now we move on to manipulating pieces of files — individual lines or groups of lines. This is another area where new users seem unsure of themselves.

Filenames

The first step is to ensure that you know the **ed** commands for reading and writing files. Of course you can't go very far without knowing **r** and **w**. Equally useful, but less well known, is the 'edit' command **e**. Within **ed**, the command

```
e newfile
```

says 'I want to edit a new file called *newfile*, without leaving the editor.' The **e** command discards whatever you're currently working on and starts over on *newfile*. It's exactly the same as if you had quit with the **q** command, then re-entered **ed** with a new file name, except that if you have a pattern remembered, then a command like // will still work.

If you enter **ed** with the command

```
ed file
```

ed remembers the name of the file, and any subsequent **e**, **r** or **w** commands that don't contain a filename will refer to this remembered file. Thus

```
ed file1
 ... (editing) ...
w       (writes back in file1)
e file2 (edit new file, without leaving editor)
 ... (editing on file2) ...
w       (writes back on file2)
```

(and so on) does a series of edits on various files without ever leaving **ed** and without typing the name of any file more than once.

You can find out the remembered file name at any time with the **f** command; just type **f** without a file name. You can also change the name of the remembered file name with **f**; a useful sequence is

```
ed  precious
f  junk
 ... (editing) ...
```

which gets a copy of a precious file, then uses **f** to guarantee that a careless **w** command won't clobber the original.

Inserting One File into Another

Suppose you have a file called 'memo', and you want the file called 'table' to be inserted just after the reference to Table 1. That is, in 'memo' somewhere is a line that says

```
Table 1 shows that ...
```

and the data contained in 'table' has to go there, probably so it will be formatted properly by **nroff** or **troff**. Now what?

This one is easy. Edit 'memo', find 'Table 1', and add the file 'table' right there:

```
ed  memo
/Table 1/
Table 1 shows that ... [response from ed]
.r  table
```

The critical line is the last one. As we said earlier, the **r** command reads a file; here you asked for it to be read in right after line dot. An **r** command without any address adds lines at the end, so it is the same as **$r**.

Writing out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to split out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited we have

```
.TS
 ...[lots of stuff]
.TE
```

which is the way a table is set up for the **tbl** program. To isolate the table in a separate file called 'table', first find the start of the table (the '.TS' line), then write out the interesting part:

```
/^\.TS/
.TS  [ed prints the line it found]
.,/^\.TE/w table
```

and the job is done. If you are confident, you can do it all at once with

```
/^\.TS/;/^\.TE/w table
```

The point is that the **w** command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. For example, if you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it — don't re-type it. In the editor, say

```
a
...lots of stuff...
...horrible line...
.
.w temp
a
...more stuff...
.
.r temp
a
...more stuff...
.
```

This last example is worth studying, to be sure you appreciate what's going on.

Moving Lines Around

Suppose you want to move a paragraph from its present position in a paper to the end. How would you do it? As a concrete example, suppose each paragraph in the paper begins with the formatting command '.PP'. Think about it and write down the details before reading on.

The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, then read in the temporary file at the end. Assuming that you are sitting on the '.PP' command that begins the paragraph, this is the sequence of commands:

```
.,/^\.PP/- w temp
.,//- d
$r temp
```

That is, from where you are now ('.') until one line before the next '.PP' ('/^\.PP/- ') write onto 'temp'. Then delete the same lines. Finally, read 'temp' at the end.

As we said, that's the brute force way. The easier way (often) is to use the *move* command **m** that **ed** provides — it lets you do the whole set of operations at one crack, without any temporary file.

The **m** command is like many other **ed** commands in that it takes up to two line numbers in front that tell what lines are to be affected. It is also *followed* by a line number that tells where the lines are to go. Thus

```
line1, line2 m line3
```

says to move all the lines between 'line1' and 'line2' after 'line3'. Naturally, any of 'line1' etc., can be patterns between slashes, $ signs, or other ways to specify lines.

Suppose again that you're sitting at the first line of the paragraph. Then you can say

```
.,/^\.PP/- m$
```

That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one to after the second. Suppose that you are positioned at the first. Then

```
m+
```

does it. It says to move line dot to after one line after line dot. If you are positioned on the second line,

```
m- -
```

does the interchange.

As you can see, the **m** command is more succinct and direct than writing, deleting and re-reading. When is brute force better anyway? This is a matter of personal taste — do what you have most confidence in. The main difficulty with the **m** command is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may well not move the lines you thought you did. The result of a botched **m** command can be a ghastly mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to issue a **w** command before doing anything complicated; then if you goof, it's easy to back up to where you were.

Marks

ed provides a facility for marking a line with a particular name so you can later reference it by name regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is **k**; the command

```
kx
```

marks the current line with the name 'x'. If a line number precedes the **k**, that line is marked. (The mark name must be a single lower case letter.) Now you can refer to the marked line with the address

```
'x
```

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with *'a*. Then find the last line and mark it with *'b*. Now position yourself at the place where the stuff is to go and say

```
'a,'bm.
```

Bear in mind that only one line can have a particular mark name associated with it at any given time.

Copying Lines

We mentioned earlier the idea of saving a line that was hard to type or used often, so as to cut down on typing time. Of course this could be more than one line; then the saving is presumably even greater.

ed provides another command, called **t** (for 'transfer') for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The **t** command is identical to the **m** command, except that instead of moving lines it simply duplicates them at the place you named. Thus

```
1,$t$
```

duplicates the entire contents that you are editing. A more common use for **t** is for creating a series of lines that differ only slightly. For example, you can say

```
a
.......... x ......... (long line)
.
t.              (make a copy)
s/x/y/          (change it a bit)
t.              (make third copy)
s/y/z/          (change it a bit)
```

and so on.

The Temporary Escape '!'

Sometimes it is convenient to be able to temporarily escape from the editor to do some other UNIX command, perhaps one of the file copy or move commands discussed in section 5, without leaving the editor. The 'escape' command **!** provides a way to do this.

If you say

```
!any- system- command
```

your current editing state is suspended, and the UNIX command you asked for is executed. When the command finishes, **ed** will signal you by printing another **!**; at that point you can resume editing.

You can really do *any* UNIX command, including another **ed**. (This is quite common, in fact.) In this case, you can even do another **!**.

7. SUPPORTING TOOLS

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how **ed** works, because they are all based on the editor. In this section we will give some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. More information on each can be found in the ROS Reference Manual (9010).

Grep

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest, but if there are many files this can get very tedious, and if the files are really big, it may be impossible because of limits in **ed**.

The program **grep** was invented to get around these limitations. The search patterns that we have described in the paper are often called 'regular expressions', and 'grep' stands for

```
g/re/p
```

That describes exactly what **grep** does — it prints every line in a set of files that contains a particular pattern. Thus

```
grep 'thing' file1 file2 file3 ...
```

finds 'thing' wherever it occurs in any of the files 'file1', 'file2', etc. **grep** also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by 'thing' can be any pattern you can use in the editor, since **grep** and **ed** use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes '...' if it contains any non-alphabetic characters, since many such characters also mean something special to the system command interpreter (the 'shell'). If you don't quote them, the command interpreter will try to interpret them before **grep** gets a chance.

There is also a way to find lines that *don't* contain a pattern:

```
grep - v 'thing' file1 file2 ...
```

finds all lines that don't contains 'thing'. The **- v** must occur in the position shown. Given **grep** and **grep - v**, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y':

```
grep x file... | grep - v y
```

(The notation | is a 'pipe', which causes the output of the first command to be used as input to the second command; see [2].)

Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a 'script', i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every 'Unix' to 'UNIX' and every 'Gcos' to 'GCOS' in a large number of files. Then put into the file 'script' the lines

```
g/Unix/s//UNIX/g
g/Gcos/s//GCOS/g
w
q
```

Now you can say

```
ed file1 <script
ed file2 <script
...
```

This causes **ed** to take its commands from the prepared script. Notice that the whole job has to be planned in advance.

And of course by using the system command interpreter, you can cycle through a set of files automatically, with varying degrees of ease.

Sed

sed ('stream editor') is a version of the editor with restricted capabilities but which is capable of processing unlimited amounts of input. Basically **sed** copies its input to its output, applying one or more editing commands to each line of input.

As an example, suppose that we want to do the 'Unix' to 'UNIX' part of the example given above, but without rewriting the files. Then the command

```
sed 's/Unix/UNIX/g' file1 file2 ...
```

applies the command 's/Unix/UNIX/g' to all lines from 'file1', 'file2', etc., and copies all lines to the output. The advantage of using **sed** in such a case is that it can be used with input too large for **ed** to handle. All the output can be collected in one place, either in a file or perhaps piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file, or on the command line, with a slightly more complex syntax. To take commands from a file, for example,

```
sed – f cmdfile input– files...
```

sed has further capabilities, including conditional testing and branching, which we cannot go into here.

Acknowledgement

Thanks to Ted Dolatta.

Ex Reference Manual

This document is based on a paper by William Joy and Mark Horton of the University of California, Berkeley. For EX version 3.5/2.13.

1. Starting ex

Each instance of the editor has a set of options, which can be set to tailor it to your liking. The command *edit* invokes a version of *ex* designed for more casual or beginning users by changing the default settings of some of these options. To simplify the description which follows we assume the default settings of the options.

When invoked, *ex* determines the terminal type from the TERM variable in the environment. It there is a TERMCAP variable in the environment, and the type of the terminal described there matches the TERM variable, then that description is used. Also if the TERMCAP variable contains a pathname (beginning with a /) then the editor will seek the description of the terminal in that file (rather than the default /etc/termcap.) If there is a variable EXINIT in the environment, then the editor will execute the commands in that variable, otherwise if there is a file *.exrc* in your HOME directory *ex* reads commands from that file, simulating a *source* command. Option setting commands placed in EXINIT or *.exrc* will be executed before each editor session.

A command to enter *ex* has the following prototype:†

ex [−] [− **v**] [− **t** *tag*] [− **r**] [− **l**] [− **w***n*] [− **x**] [− **R**] [+*command*] name ...

The most common case edits a single file with no options, i.e.:

ex name

The − command line option option suppresses all interactive-user feedback and is useful in processing editor scripts in command files. The − **v** option is equivalent to using *vi* rather than *ex*. The − **t** option is equivalent to an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition. The − **r** option is used in recovering after an editor or system crash, retrieving the last saved version of the named file or, if no file is specified, typing a list of saved files. The − **l** option sets up for editing LISP, setting the *showmatch* and *lisp* options. The − **w** option sets the default window size to *n*, and is useful on dialups to start in small windows. The − **x** option causes *ex* to prompt for a *key*, which is used to encrypt and decrypt the contents of the file, which should already be encrypted using the same key, see *crypt*(1). The − **R** option sets the *readonly* option at the start. ‡ *Name* arguments indicate files to be edited. An argument of the form +*command* indicates that the editor should begin by executing the specified command. If *command* is omitted, then it defaults to "$", positioning the editor at the last line of the first file initially. Other useful commands here are scanning patterns of the form "/pat" or line numbers, e.g. "+100" starting at line 100.

2. File manipulation

2.1. Current file

Ex is normally editing the contents of a single file, whose name is recorded in the *current* file name. *Ex* performs all editing actions in a buffer (actually a temporary file) into which the text of the file is initially read. Changes made to the buffer have no effect on the file being edited unless and until the buffer contents are written out to the file with a *write* command.

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

† Brackets '[' ']' surround optional parameters here.

‡ Not available in all v2 editors due to memory constraints.

After the buffer contents are written, the previous contents of the written file are no longer accessible. When a file is edited, its name becomes the current file name, and its contents are read into the buffer.

The current file is almost always considered to be *edited.* This means that the contents of the buffer are logically connected with the current file name, so that writing the current buffer contents onto that file, even if it exists, is a reasonable action. If the current file is not *edited* then *ex* will not normally write on it if it already exists.*

2.2. Alternate file

Each time a new value is given to the current file name, the previous current file name is saved as the *alternate* file name. Similarly if a file is mentioned but does not become the current file, it is saved as the alternate file name.

2.3. Filename expansion

Filenames within the editor may be specified using the normal shell expansion conventions. In addition, the character '%' in filenames is replaced by the *current* file name and the character '#' by the *alternate* file name.†

2.4. Multiple files and named buffers

If more than one file is given on the command line, then the first file is edited as described above. The remaining arguments are placed with the first file in the *argument list.* The current argument list may be displayed with the *args* command. The next file in the argument list may be edited with the *next* command. The argument list may also be respecified by specifying a list of names to the *next* command. These names are expanded, the resulting list of names becomes the new argument list, and *ex* edits the first file on the list.

For saving blocks of text while editing, and especially when editing more than one file, *ex* has a group of named buffers. These are similar to the normal buffer, except that only a limited number of operations are available on them. The buffers have names *a* through *z*.‡

2.5. Read only

It is possible to use *ex* in *read only* mode to look at files that you have no intention of modifying. This mode protects you from accidently overwriting the file. Read only mode is on when the *readonly* option is set. It can be turned on with the **–R** command line option, by the *view* command line invocation, or by setting the *readonly* option. It can be cleared by setting *noreadonly.* It is possible to write, even while in read only mode, by indicating that you really know what you are doing. You can write to a different file, or can use the ! form of write, even while in read only mode.

3. Exceptional Conditions

3.1. Errors and interrupts

When errors occur *ex* (optionally) rings the terminal bell and, in any case, prints an error diagnostic. If the primary input is from a file, editor processing will terminate. If an interrupt signal is received, *ex* prints "Interrupt" and returns to its command level. If the primary input is a file, then *ex* will exit when this occurs.

* The *file* command will say "[Not edited]" if the current file is not considered edited.

† This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an *edit* command after a *No write since last change* diagnostic is received.

‡ It is also possible to refer to *A* through *Z;* the upper case buffers are the same as the lower but commands append to named buffers rather than replacing if upper case names are used.

3.2. Recovering from hangups and crashes

If a hangup signal is received and the buffer has been modified since it was last written out, or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second) will attempt to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing at most a few lines of changes from the last point before the hangup or editor crash. To recover a file you can use the **– r** option. If you were editing the file *resume,* then you should change to the directory where you were when the crash occurred, giving the command

ex – r *resume*

After checking that the retrieved file is indeed ok, you can *write* it over the previous contents of that file.

You will normally get mail from the system telling you when a file has been saved after a crash. The command

ex – r

will print a list of the files which have been saved for you. (In the case of a hangup, the file will not appear in the list, although it can be recovered.)

4. Editing modes

Ex has five distinct modes. The primary mode is *command* mode. Commands are entered in command mode when a ':' prompt is present, and are executed each time a complete line is sent. In *text input* mode *ex* gathers input lines and places them in the file. The *append, insert,* and *change* commands use text input mode. No prompt is printed when you are in text input mode. This mode is left by typing a '.' alone at the beginning of a line, and *command* mode resumes.

The last three modes are *open* and *visual* modes, entered by the commands of the same name, and, within open and visual modes *text insertion* mode. *Open* and *visual* modes allow local editing operations to be performed on the text in the file. The *open* command displays one line at a time on any terminal while *visual* works on CRT terminals with random positioning cursors, using the screen as a (single) window for file editing changes. These modes are described (only) in *An Introduction to Display Editing with Vi.*

5. Command structure

Most command names are English words, and initial prefixes of the words are acceptable abbreviations. The ambiguity of abbreviations is resolved in favor of the more commonly used commands.*

5.1. Command parameters

Most commands accept prefix addresses specifying the lines in the file upon which they are to have effect. The forms of these addresses will be discussed below. A number of commands also may take a trailing *count* specifying the number of lines to be involved in the command.† Thus the command "10p" will print the tenth line in the buffer while "delete 5" will delete five lines from the buffer, starting with the current line.

Some commands take other information or parameters, this information always being given after the command name.‡

---

* As an example, the command *substitute* can be abbreviated 's' while the shortest available abbreviation for the *set* command is 'se'.

† Counts are rounded down if necessary.

‡ Examples would be option names in a *set* command i.e. "set number", a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command, i.e. "1,5 copy 25".

5.2. Command variants

A number of commands have two distinct variants. The variant form of the command is invoked by placing an '!' immediately after the command name. Some of the default variants may be controlled by options; in this case, the '!' serves to toggle the default.

5.3. Flags after commands

The characters '#', 'p' and 'l' may be placed after many commands.** In this case, the command abbreviated by these characters is executed after the command completes. Since *ex* normally prints the new current line after each change, 'p' is rarely necessary. Any number of '+ ' or '– ' characters may also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

5.4. Comments

It is possible to give editor commands which are ignored. This is useful when making complex editor scripts for which comments are desired. The comment character is the double quote: ". Any command line beginning with " is ignored. Comments beginning with " may also be placed at the ends of commands, except in cases where they could be confused as part of text (shell escapes and the substitute and map commands).

5.5. Multiple commands per line

More than one command may be placed on a line by separating each pair of commands by a '|' character. However the *global* commands, comments, and the shell escape '!' must be the last command on a line, as they are not terminated by a '|'.

5.6. Reporting large changes

Most commands which change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the *report* option. This feedback helps to detect undesirably large changes so that they may be quickly and easily reversed with an *undo*. After commands with more global effect such as *global* or *visual*, you will be informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

6. Command addressing

6.1. Addressing primitives

.	The current line. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, thus '.' is rarely used alone as an address.
n	The *n*th line in the editor's buffer, lines being numbered sequentially from 1.
$	The last line in the buffer.
%	An abbreviation for "1,$", the entire buffer.
+n – n	An offset relative to the current buffer line.†
/pat/ ?pat?	Scan forward and backward respectively for a line containing *pat*, a regular expression (as defined below). The scans normally wrap around the end of the buffer. If all that is desired is to print the next line containing *pat*, then the trailing / or ? may be omitted. If *pat* is omitted or explicitly empty, then the last regular expression specified is located.‡

** A 'p' or 'l' must be preceded by a blank or tab except in the single special case 'dp'.

† The forms '.+3' '+3' and '+++' are all equivalent; if the current line is line 100 they all address line 103.

‡ The forms \/ and \? scan using the last regular expression used in a scan; after a substitute // and ??

'' 'x Before each non-relative motion of the current line '.', the previous current line is marked with a tag, subsequently referred to as ''''. This makes it easy to refer or return to this previous context. Marks may also be established by the *mark* command, using single lower case letters *x* and the marked lines referred to as '*'x*'.

6.2. Combining addressing primitives

Addresses to commands consist of a series of addressing primitives, separated by ',' or ';'. Such address lists are evaluated left-to-right. When addresses are separated by ';' the current line '.' is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer.†

7. Command descriptions

The following form is a prototype for all *ex* commands:

address **command** *! parameters count flags*

All parts are optional; the degenerate case is the empty command which prints the next line in the file. For sanity with use from within *visual* mode, *ex* ignores a ":" preceding any command.

In the following command descriptions, the default addresses are shown in parentheses, which are *not,* however, part of the command.

abbreviate *word rhs* abbr: **ab**

Add the named abbreviation to the current list. When in input mode in visual, if *word* is typed as a complete word, it will be changed to *rhs.*

(.) **append** abbr: **a**
text
.

Reads the input text and places it after the specified line. After the command, '.' addresses the last line input or the specified line if no lines were input. If address '0' is given, text is placed at the beginning of the buffer.

a!
text
.

The variant flag to *append* toggles the setting for the *autoindent* option during the input of *text.*

args

The members of the argument list are printed, with the current argument delimited by '[' and ']'.

would scan using the substitute's regular expression.
† Null address specifications are permitted in a list of addresses, the default in this case is the current line '.'; thus ',100' is equivalent to '.,100'. It is an error to give a prefix address to a command which expects none.

(. , .) **change** *count* abbr: **c**
text
.

Replaces the specified lines with the input *text*. The current line becomes the last line input; if no lines were input it is left as for a *delete*.

c!
text
.

The variant toggles *autoindent* during the *change*.

(. , .) **copy** *addr flags* abbr: **co**

A *copy* of the specified lines is placed after *addr*, which may be '0'. The current line '.' addresses the last line of the copy. The command *t* is a synonym for *copy*.

(. , .) **delete** *buffer count flags* abbr: **d**

Removes the specified lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line. If a named *buffer* is specified by giving a letter, then the specified lines are saved in that buffer, or appended to it if an upper case letter is used.

edit *file* abbr: **e**
ex *file*

Used to begin an editing session on a new file. The editor first checks to see if the buffer has been modified since the last *write* command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After insuring that this file is sensible† the editor reads the file into its buffer.

If the read of the file completes without error, the number of lines and characters read is typed. If there were any non-ASCII characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurred, the file is considered *edited*. If the last line of the input file is missing the trailing newline character, it will be supplied and a complaint will be issued. This command leaves the current line '.' at the last line read.‡

e! *file*

The variant form suppresses the complaint about modifications having been made and not written from the editor buffer, thus discarding all changes which have been made before editing the new file.

e +*n* *file*

Causes the editor to begin at line *n* rather than at the last line; *n* may also be an editor command containing no spaces, e.g.: "+ /pat".

file abbr: **f**

† I.e., that it is not a binary file such as a directory, a block or character special file other than */dev/tty*, a terminal, or a binary or executable file (as indicated by the first word).
‡ If executed from within *open* or *visual*, the current line is initially the first line of the file.

Prints the current file name, whether it has been '[Modified]' since the last *write* command, whether it is *read only*, the current line, the number of lines in the buffer, and the percentage of the way through the buffer of the current line.*

file *file*

The current file name is changed to *file* which is considered '[Not edited]'.

(1 , $) **global** */pat/ cmds* abbr: **g**

First marks each line among those specified which matches the given regular expression. Then the given command list is executed with '.' initially set to each marked line.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a '\'. If *cmds* (and possibly the trailing / delimiter) is omitted, each line matching *pat* is printed. *Append, insert,* and *change* commands and associated input are permitted; the '.' terminating input may be omitted if it would be on the last line of the command list. *Open* and *visual* commands are permitted in the command list and take input from the terminal.

The *global* command itself may not appear in *cmds*. The *undo* command is also not permitted there, as *undo* instead can be used to reverse the entire *global* command. The options *autoprint* and *autoindent* are inhibited during a *global*, (and possibly the trailing / delimiter) and the value of the *report* option is temporarily infinite, in deference to a *report* for the entire global. Finally, the context mark '´´' is set to the value of '.' before the global command begins and is not changed during a global command, except perhaps by an *open* or *visual* within the *global*.

g! */pat/ cmds* abbr: **v**

The variant form of *global* runs *cmds* at each line not matching *pat*.

(.) **insert** abbr: **i**
text
.

Places the given text before the specified line. The current line is left at the last line input; if there were none input it is left at the line before the addressed line. This command differs from *append* only in the placement of text.

i!
text
.

The variant toggles *autoindent* during the *insert*.

(. , .+ 1) **join** *count flags* abbr: **j**

Places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character, two if there was a '.' at the end of the line, or none if the first following character is a ')'. If there is already white space at the end of the line, then the white space at the start of the next line will be discarded.

* In the rare case that the current file is '[Not edited]' this is noted also; in this case you have to use the form **w!** to write to the file, since the editor is not sure that a **write** will not destroy a file unrelated to the current contents of the buffer.

j!

The variant causes a simpler *join* with no white space processing; the characters in the lines are simply concatenated.

(.) **k** *x*

The *k* command is a synonym for *mark*. It does not require a blank or tab before the following letter.

(. , .) **list** *count flags*

Prints the specified lines in a more unambiguous way: tabs are printed as '^I' and the end of each line is marked with a trailing '$'. The current line is left at the last line printed.

map *lhs rhs*

The *map* command is used to define macros for use in *visual* mode. *Lhs* should be a single character, or the sequence "#n", for n a digit, referring to function key *n*. When this character or function key is typed in *visual* mode, it will be as though the corresponding *rhs* had been typed. On terminals without function keys, you can type "#n". See section 6.9 of the "Introduction to Display Editing with Vi" for more details.

(.) **mark** *x*

Gives the specified line mark *x*, a single lower case letter. The *x* must be preceded by a blank or a tab. The addressing form '`x' then addresses this line. The current line is not affected by this command.

(. , .) **move** *addr* abbr: **m**

The *move* command repositions the specified lines to be after *addr*. The first of the moved lines becomes the current line.

next abbr: **n**

The next file from the command line argument list is edited.

n!

The variant suppresses warnings about the modifications to the buffer not having been written out, discarding (irretrievably) any changes which may have been made.

n *filelist*
n + *command filelist*

The specified *filelist* is expanded and the resulting list replaces the current argument list; the first file in the new list is then edited. If *command* is given (it must contain no spaces), then it is executed after editing the first such file.

(. , .) **number** *count flags* abbr: **#** or **nu**

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

(.) **open** *flags* abbr: **o**
(.) **open** */pat/ flags*

Enters intraline editing *open* mode at each addressed line. If *pat* is given, then the cursor will be placed initially at the beginning of the string matched by the pattern. To exit this mode use Q. See *An Introduction to Display Editing with Vi* for more details.
‡

‡ Not available in all v2 editors due to memory constraints.

preserve

The current editor buffer is saved as though the system had just crashed. This command is for use only in emergencies when a *write* command has resulted in an error and you don't know how to save your work. After a *preserve* you should seek help.

(. , .) **print** *count* abbr: **p** or **P**

Prints the specified lines with non-printing characters printed as control characters '^*x*'; delete (octal 177) is represented as '^?'. The current line is left at the last line printed.

(.) **put** *buffer* abbr: **pu**

Puts back previously *deleted* or *yanked* lines. Normally used with *delete* to effect movement of lines, or with *yank* to effect duplication of lines. If no *buffer* is specified, then the last *deleted* or *yanked* text is restored.* By using a named buffer, text may be restored that was saved there at any previous time.

quit abbr: **q**

Causes *ex* to terminate. No automatic write of the editor buffer to a file is performed. However, *ex* issues a warning message if the file has changed since the last *write* command was issued, and does not *quit*.† Normally, you will wish to save your changes, and you should give a *write* command; if you wish to discard them, use the **q!** command variant.

q!

Quits from the editor, discarding changes to the buffer without complaint.

(.) **read** *file* abbr: **r**

Places a copy of the text of the given file in the editing buffer after the specified line. If no *file* is given the current file name is used. The current file name is not changed unless there is none in which case *file* becomes the current name. The sensibility restrictions for the *edit* command apply here also. If the file buffer is empty and there is no current name then *ex* treats this as an *edit* command.

Address '0' is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the *edit* command when the *read* successfully terminates. After a *read* the current line is the last line read.‡

(.) **read** !*command*

Reads the output of the command *command* into the buffer after the specified line. This is not a variant form of the command, rather a read specifying a *command* rather than a *filename;* a blank or tab before the ! is mandatory.

recover *file*

Recovers *file* from the system save area. Used after a accidental hangup of the phone** or a system crash** or *preserve* command. Except when you use *preserve* you will be notified by mail when a file is saved.

* But no modifying commands may intervene between the *delete* or *yank* and the *put,* nor may lines be moved between files without using a named buffer.
† *Ex* will also issue a diagnostic if there are more files in the argument list.
‡ Within *open* and *visual* the current line is set to the first line read rather than the last.
** The system saves a copy of the file you were editing only if you have made changes to the file.

rewind abbr: **rew**

The argument list is rewound, and the first file in the list is edited.

rew!

Rewinds the argument list discarding any changes made to the current buffer.

set *parameter*

With no arguments, prints those options whose values have been changed from their defaults; with parameter *all* it prints all of the option values.

Giving an option name followed by a '?' causes the current value of that option to be printed. The '?' is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form 'set *option*' to turn them on or 'set no*option*' to turn them off; string and numeric options are assigned via the form 'set *option*=value'.

More than one parameter may be given to *set*; they are interpreted left-to-right.

shell abbr: **sh**

A new shell is created. When it terminates, editing resumes.

source *file* abbr: **so**

Reads and executes commands from the specified file. *Source* commands may be nested.

(. , .) **substitute** */pat/repl/ options count flags* abbr: **s**

On each specified line, the first instance of pattern *pat* is replaced by replacement pattern *repl.* If the *global* indicator option character 'g' appears, then all instances are substituted; if the *confirm* indication character 'c' appears, then before each substitution the line to be substituted is typed with the string to be substituted marked with '↑' characters. By typing an 'y' one can cause the substitution to be performed, any other input causes no change to take place. After a *substitute* the current line is the last line substituted.

Lines may be split by substituting new-line characters into them. The newline in *repl* must be escaped by preceding it with a '\'. Other metacharacters available in *pat* and *repl* are described below.

stop

Suspends the editor, returning control to the top level shell. If *autowrite* is set and there are unsaved changes, a write is done first unless the form **stop**! is used. This commands is only available where supported by the teletype driver and operating system.

(. , .) **substitute** *options count flags* abbr: **s**

If *pat* and *repl* are omitted, then the last substitution is repeated. This is a synonym for the **&** command.

(. , .) **t** *addr flags*

The *t* command is a synonym for *copy.*

ta *tag*

The focus of editing switches to the location of *tag,* switching to a different line in the current file where it is defined, or if necessary to another file.‡

‡ If you have modified the current file before giving a *tag* command, you must write it out; giving another *tag* command, specifying no *tag* will reuse the previous tag.

The tags file is normally created by a program such as *ctags*, and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form which can be used by the editor to find the tag; this field is usually a contextual scan using '*/pat/*' to be immune to minor changes in the file. Such scans are always performed as if *nomagic* was set.

The tag names in the tags file must be sorted alphabetically. ‡

unabbreviate *word* abbr: **una**

Delete *word* from the list of abbreviations.

undo abbr: **u**

Reverses the changes made in the buffer by the last buffer editing command. Note that *global* commands are considered a single command for the purpose of *undo* (as are *open* and *visual.)* Also, the commands *write* and *edit* which interact with the file system cannot be undone. *Undo* is its own inverse.

Undo always marks the previous value of the current line '**.**' as '´´'. After an *undo* the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect such as *global* and *visual* the current line regains it's pre-command value after an *undo.*

unmap *lhs*

The macro expansion associated by *map* for *lhs* is removed.

(1 , $) **v** */pat/ cmds*

A synonym for the *global* command variant **g!**, running the specified *cmds* on each line which does not match *pat.*

version abbr: **ve**

Prints the current version number of the editor as well as the date the editor was last changed.

(**.**) **visual** *type count flags* abbr: **vi**

Enters visual mode at the specified line. *Type* is optional and may be '– ' , '↑' or '**.**' as in the *z* command to specify the placement of the specified line on the screen. By default, if *type* is omitted, the specified line is placed as the first on the screen. A *count* specifies an initial window size; the default is the value of the option *window.* See the document *An Introduction to Display Editing with Vi* for more details. To exit this mode, type Q.

visual file
visual + *n* file

From visual mode, this command is the same as edit.

(1 , $) **write** *file* abbr: **w**

Writes changes made back to *file,* printing the number of lines and characters written. Normally *file* is omitted and the text goes back where it came from. If a *file* is specified, then text will be written to that file.* If the file does not exist it is created. The current file name is changed only if there is no current file name; the current line is never

‡ Not available in all v2 editors due to memory constraints.
* The editor writes to a file only if it is the current file and is *edited,* if the file does not exist, or if the file is actually a teletype, */dev/tty, /dev/null.* Otherwise, you must give the variant form **w!** to force the write.

changed.

If an error occurs while writing the current and *edited* file, the editor considers that there has been "No write since last change" even if the buffer had not previously been modified.

(1 , $) **write>>** *file* abbr: **w>>**

Writes the buffer contents at the end of an existing file.

w! *name*

Overrides the checking of the normal *write* command, and will write to any file which the system permits.

(1 , $) **w !***command*

Writes the specified lines into *command.* Note the difference between **w!** which overrides checks and **w !** which writes to a command.

wq *name*

Like a *write* and then a *quit* command.

wq! *name*

The variant overrides checking on the sensibility of the *write* command, as **w!** does.

xit *name*

If any changes have been made and not written, writes the buffer out. Then, in any case, quits.

(. , .) **yank** *buffer count* abbr: **ya**

Places the specified lines in the named *buffer,* for later retrieval via *put.* If no buffer name is specified, the lines go to a more volatile place; see the *put* command description.

(.+1) **z** *count*

Print the next *count* lines, default *window.*

(.) **z** *type count*

Prints a window of text with the specified line at the top. If *type* is '– ' the line is placed at the bottom; a '.' causes the line to be placed in the center.* A count gives the number of lines to be displayed rather than double the number specified by the *scroll* option. On a CRT the screen is cleared before display begins unless a count which is less than the screen size is given. The current line is left at the last line printed.

! *command*

The remainder of the line after the '!' character is sent to a shell to be executed. Within the text of *command* the characters '%' and '#' are expanded as in filenames and the character '!' is replaced with the text of the previous command. Thus, in particular, '!!' repeats the last such shell escape. If any such expansion is performed, the expanded line will be echoed. The current line is unchanged by this command.

* Forms 'z=' and 'z↑' also exist; 'z=' places the current line in the center, surrounds it with lines of '– ' characters and leaves the current line at this line. The form 'z↑' prints the window before 'z– ' would. The characters '+', '↑' and '– ' may be repeated for cumulative effect. On some v2 editors, no *type* may be given.

If there has been "[No write]" of the buffer contents since the last change to the editing buffer, then a diagnostic will be printed before the command is executed as a warning. A single '!' is printed when the command completes.

(*addr* , *addr*) **!** *command*

Takes the specified address range and supplies it as standard input to *command;* the resulting output then replaces the input lines.

($) =

Prints the line number of the addressed line. The current line is unchanged.

(. , .) > *count flags*
(. , .) < *count flags*

Perform intelligent shifting on the specified lines; < shifts left and > shift right. The quantity of shift is determined by the *shiftwidth* option and the repetition of the specification character. Only white space (blanks and tabs) is shifted; no non-white characters are discarded in a left-shift. The current line becomes the last line which changed due to the shifting.

^D

An end-of-file from a terminal input scrolls through the file. The *scroll* option specifies the size of the scroll, normally a half screen of text.

(.+1 , .+1)
(.+1 , .+1) |

An address alone causes the addressed lines to be printed. A blank line prints the next line in the file.

(. , .) & *options count flags*

Repeats the previous *substitute* command.

(. , .) ~ *options count flags*

Replaces the previous regular expression with the previous replacement pattern from a substitution.

8. Regular expressions and substitute replacement patterns

8.1. Regular expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. *Ex* remembers two previous regular expressions: the previous regular expression used in a *substitute* command and the previous regular expression used elsewhere (referred to as the previous *scanning* regular expression.) The previous regular expression can always be referred to by a null *re,* e.g. '//' or '??'.

8.2. Magic and nomagic

The regular expressions allowed by *ex* are constructed in one of two ways depending on the setting of the *magic* option. The *ex* and *vi* default setting of *magic* gives quick access to a powerful set of regular expression metacharacters. The disadvantage of *magic* is that the user must remember that these metacharacters are *magic* and precede them with the character '\' to use them as "ordinary" characters. With *nomagic,* the default for *edit,* regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the (now) ordinary character with a '\'. Note that '\' is thus always a metacharacter.

The remainder of the discussion of regular expressions assumes that that the setting of this option is *magic*.†

8.3. Basic regular expression summary

The following basic constructs are used to construct *magic* mode regular expressions.

char	An ordinary character matches itself. The characters '↑' at the beginning of a line, '$' at the end of line, '*' as any character other than the first, '.', '\', '[', and '~' are not ordinary characters and must be escaped (preceded) by '\' to be treated as such.
↑	At the beginning of a pattern forces the match to succeed only at the beginning of a line.
$	At the end of a regular expression forces the match to succeed only at the end of the line.
.	Matches any single character except the new-line character.
\<	Forces the match to occur only at the beginning of a "variable" or "word"; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.
\>	Similar to '\<', but matching the end of a "variable" or "word", i.e. either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.
[*string*]	Matches any (single) character in the class defined by *string*. Most characters in *string* define themselves. A pair of characters separated by '– ' in *string* defines the set of characters collating between the specified lower and upper bounds, thus '[a– z]' as a regular expression matches any (single) lower-case letter. If the first character of *string* is an '↑' then the construct matches those characters which it otherwise would not; thus '[↑a– z]' matches anything but a lower-case letter (and of course a newline). To place any of the characters '↑', '[', or '– ' in *string* you must escape them with a preceding '\'.

8.4. Combining regular expression primitives

The concatenation of two regular expressions matches the leftmost and then longest string which can be divided with the first piece matching the first regular expression and the second piece matching the second. Any of the (single character matching) regular expressions mentioned above may be followed by the character '*' to form a regular expression which matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows.

The character '~' may be used in a regular expression, and matches the text which defined the replacement part of the last *substitute* command. A regular expression may be enclosed between the sequences '\(' and '\)' with side effects in the *substitute* replacement patterns.

8.5. Substitute replacement patterns

The basic metacharacters for the replacement pattern are '&' and '~'; these are given as '\&' and '\~' when *nomagic* is set. Each instance of '&' is replaced by the characters which the regular expression matched. The metacharacter '~' stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by the escaping character '\'. The sequence '*n*' is replaced by the text matched by the *n*-th regular

† To discern what is true with *nomagic* it suffices to remember that the only special characters in this case will be '↑' at the beginning of a regular expression, '$' at the end of a regular expression, and '\'. With *nomagic* the characters '~' and '&' also lose their special meanings related to the replacement pattern of a substitute.

subexpression enclosed between '\(' and '\)'.† The sequences '\u' and '\l' cause the immediately following character in the replacement to be converted to upper- or lower-case respectively if this character is a letter. The sequences '\U' and '\L' turn such conversion on, either until '\E' or '\e' is encountered, or until the end of the replacement pattern.

9. Option descriptions

autoindent, ai default: noai

Can be used to ease the preparation of structured program text. At the beginning of each *append, change* or *insert* command or when a new line is *opened* or created by an *append, change, insert,* or *substitute* operation within *open* or *visual* mode, *ex* looks at the line being appended after, the first line changed or the line inserted before and calculates the amount of white space at the start of the line. It then aligns the cursor at the level of indentation so determined.

If the user then types lines of text in, they will continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line will start aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop one can hit **^D**. The tab stops going backwards are defined at multiples of the *shiftwidth* option. You *cannot* backspace over the indent, except by sending an end-of-file with a **^D**.

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the *autoindent* is discarded.) Also specially processed in this mode are lines beginning with an '↑' and immediately followed by a **^D**. This causes the input to be repositioned at the beginning of the line, but retaining the previous indent for the next line. Similarly, a '0' followed by a **^D** repositions at the beginning but without retaining the previous indent.

Autoindent doesn't happen in *global* commands or when the input is not a terminal.

autoprint, ap default: ap

Causes the current line to be printed after each *delete, copy, join, move, substitute, t, undo* or shift command. This has the same effect as supplying a trailing 'p' to each such command. *Autoprint* is suppressed in globals, and only applies to the last of many commands on a line.

autowrite, aw default: noaw

Causes the contents of the buffer to be written to the current file if you have modified it and give a *next, rewind, stop, tag,* or *!* command, or a **^↑** (switch files) or ^] (tag goto) command in *visual.* Note, that the *edit* and *ex* commands do **not** autowrite. In each case, there is an equivalent way of switching when autowrite is set to avoid the *autowrite* (*edit* for *next, rewind!* for .I rewind , *stop!* for *stop, tag!* for *tag, shell* for *!,* and **:e #** and a **:ta!** command from within *visual).*

beautify, bf default: nobeautify

Causes all control characters except tab, newline and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. *Beautify* does not apply to command input.

† When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of '\(' starting from the left.

directory, dir default: dir=/tmp

Specifies the directory in which *ex* places its buffer file. If this directory in not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edcompatible default: noedcompatible

Causes the presence of absence of **g** and **c** suffixes on substitute commands to be remembered, and to be toggled by repeating the suffices. The suffix **r** makes the substitution be as in the ~ command, instead of like *&*. ‡‡

errorbells, eb default: noeb

Error messages are preceded by a bell.* If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht default: ht=8

Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

ignorecase, ic default: noic

All upper case characters in the text are mapped to lower case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.

lisp default: nolisp

Autoindent indents appropriately for *lisp* code, and the **( ) { } [[** and **]]** commands in *open* and *visual* are modified to have meaning for *lisp*.

list default: nolist

All printed lines will be displayed (more) unambiguously, showing tabs and end-of-lines as in the *list* command.

magic default: magic for *ex* and *vi*†

If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only '↑' and '$' having special effects. In addition the metacharacters '~' and '&' of the replacement pattern are treated as normal characters. All the normal metacharacters may be made *magic* when *nomagic* is set by preceding them with a '\'.

mesg default: mesg

Causes write permission to be turned off to the terminal while you are in visual mode, if *nomesg* is set. ‡‡

number, nu default: nonumber

Causes all output lines to be printed with their line numbers. In addition each input line will be prompted for by supplying the line number it will have.

‡‡ Version 3 only.
* Bell ringing in *open* and *visual* on errors is not suppressed by setting *noeb*.
† *Nomagic* for *edit*.
‡‡ Version 3 only.

open default: open

If *noopen*, the commands *open* and *visual* are not permitted. This is set for *edit* to prevent confusion resulting from accidental entry to open or visual mode.

optimize, opt default: optimize

Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

paragraphs, para default: para=IPLPPPQPP LIbp

Specifies the paragraphs for the { and } operations in *open* and *visual*. The pairs of characters in the option's value are the names of the macros which start paragraphs.

prompt default: prompt

Command mode input is prompted for with a ':'.

redraw default: noredraw

The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal (e.g. during insertions in *visual* the characters to the right of the cursor position are refreshed as each input character is typed.) Useful only at very high speed.

remap default: remap

If on, macros are repeatedly tried until they are unchanged. ‡‡ For example, if **o** is mapped to **O**, and **O** is mapped to **I**, then if *remap* is set, **o** will map to **I**, but if *noremap* is set, it will map to **O**.

report default: report=5†

Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as *global, open, undo,* and *visual* which have potentially more far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a *global* command on the individual commands performed.

scroll default: scroll= window

Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode *z* command (double the value of *scroll*).

sections default: sections=SHNHH HU

Specifies the section macros for the [[and]] operations in *open* and *visual*. The pairs of characters in the options's value are the names of the macros which start paragraphs.

shell, sh default: sh=/bin/sh

Gives the path name of the shell forked for the shell escape command '!', and by the *shell* command. The default is taken from SHELL in the environment, if present.

‡‡ Version 3 only.
† 2 for *edit*.

shiftwidth, sw default: sw=8

Gives the width a software tab stop, used in reverse tabbing with **^D** when using *autoindent* to append text, and by the shift commands.

showmatch, sm default: nosm

In *open* and *visual* mode, when a **)** or **}** is typed, move the cursor to the matching **(** or **{** for one second if this matching character is on the screen. Extremely useful with *lisp*.

slowopen, slow terminal dependent

Affects the display algorithm used in *visual* mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent. See *An Introduction to Display Editing with Vi* for more details.

tabstop, ts default: ts=8

The editor expands tabs in the input file to be on *tabstop* boundaries for the purposes of display.

taglength, tl default: tl=0

Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

tags default: tags=tags /usr/lib/tags

A path of files to be used as tag files for the *tag* command. ‡‡ A requested tag is searched for in the specified files, sequentially. By default (even in version 2) files called **tags** are searched for in the current directory and in /usr/lib (a master file for the entire system.)

term from environment TERM

The terminal type of the output device.

terse default: noterse

Shorter error diagnostics are produced for the experienced user.

warn default: warn

Warn if there has been '[No write since last change]' before a '!' command escape.

window default: window=speed dependent

The number of lines in a text window in the *visual* command. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

w300, w1200, w9600

These are not true options but set **window** only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.

wrapscan, ws default: ws

Searches using the regular expressions in addressing will wrap around past the end of the file.

‡‡ Version 3 only.

wrapmargin, wm default: wm=0

Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. See *An Introduction to Display Editing with Vi* for details.

writeany, wa default: nowa

Inhibit the checks normally made before *write* commands, allowing a write to any file which the system protection mechanism will allow.

10. Limitations

Editor limits that the user is likely to encounter are as follows: 1024 characters per line, 256 characters per global command list, 128 characters per file name, 128 characters in the previous inserted and deleted text in *open* or *visual,* 100 characters in a shell escape command, 63 characters in a string valued option, and 30 characters in a tag name, and a limit of 250000 lines in the file is silently enforced.

The *visual* implementation limits the number of macros defined with map to 32, and the total number of characters in macros to be less than 512.

Acknowledgments. Chuck Haley contributed greatly to the early development of *ex*. Bruce Englar encouraged the redesign which led to *ex* version 1. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and Unix systems.

Edit: A Tutorial

This document is based on a paper by Ricki Blau and James Joyce of the University of California, Berkeley.

Edit is a more powerful version of an editor called EX.

These lessons assume no prior familiarity with computers or with text editing. They consist of a series of text editing sessions which will lead you through the fundamental steps of creating and revising a file of text. After scanning each lesson and before beginning the next, you should follow the examples at a terminal to get a feeling for the actual process of text editing. Set aside some time for experimentation, and you will soon become familiar with using the computer to write and modify text. In addition to the actual use of the text editor, other features of ROS will be very important to your work. You can begin to learn about these other features by reading a tutorial introduction to the system. You will be ready to proceed with this lesson as soon as you are familiar with your terminal and its special keys, the login procedure, and the ways of correcting typing errors. Let's first define some terms:

program	A set of instructions given to the computer, describing the sequence of steps which the computer performs in order to accomplish a specific task. As an example, a series of steps to balance your checkbook is a program.
ROS	ROS is a special type of program, called an operating system, that supervises the machinery and all other programs comprising the total computer system.
edit	*edit* is the name of a text editor which you will be learning to use, a program that aids you in writing or revising text. Edit was designed for beginning users, and is a simplified version of an editor called *ex*.
file	Each account is allotted space for the permanent storage of information, such as programs, data or text. A file is a logical unit of data, for example, an essay, a program, or a chapter from a book, which is stored on a computer system. Once you create a file, it is kept until you instruct the system to remove it. You may create a file during one computer session, log out, and return to use it at a later time. Files contain anything you choose to write and store in them. The sizes of files vary to suit your needs; one file might hold only a single number while another might contain a very long document or program. The only way to save information from one session to the next is to store it in a file.
filename	Filenames are used to distinguish one file from another, serving the same purpose as the labels of manila folders in a file cabinet. In order to write or access information in a file, you use the name of that file in a ROS command, and the system will automatically locate the file.
disk	Files are stored on an input/output device called a disk, which looks something like a stack of phonograph records. Each surface is coated with a material similar to the coating on magnetic recording tape, on which information is recorded.
buffer	A temporary work space, made available to the user for the duration of a session of text editing and used for building and modifying the text file. We can imagine the buffer as a blackboard that is erased after each class, where each session with the editor is a class.

Session 1: Creating a File of Text

To use the editor you must first make contact with the computer by logging in to ROS. We'll quickly review the standard ROS login procedure.

If the terminal you are using is directly linked to the computer, turn it on and press carriage return, usually labeled "RETURN". If your terminal connects with the computer over a telephone line, turn on the terminal, dial the system access number, and, when you hear a high-pitched tone, place the receiver of the telephone in the acoustic coupler. Press carriage return once and await the login message:

```
:login:
```

Type your login name, which identifies you to the system, on the same line as the login message, and press return. If the terminal you are using has both upper and lower case, enter your login name in lower case; otherwise ROS assumes your terminal has only upper case and will not recognize lower case letters later in the session. ROS displays ":login:" and you reply with your login name, for example "susan":

:login: **susan** *(and press carriage return)*

(In the examples, input typed by the user appears in **bold face** to distinguish it from the system responses.)

The system now requests a password as an additional precaution to prevent unauthorized people from using your account. To prevent other people from seeing your password, it will not appear when you type it. The message is:

Password: *(type your password and press carriage return)*

If any of the information you gave during the login sequence was mistyped or incorrect, the system responds

```
Login incorrect.
:login:
```

in which case you should start the login process anew. Assuming that you have successfully logged in, a welcome message is displayed and eventually the $ prompt appears on a new line. The $ is the ROS symbol which tells you that it is ready to accept a command.

Asking for *edit*

You are ready to tell ROS that you want to work with edit, the text editor. Now is a convenient time to choose a name for the file of text which you are about to create. To begin your editing session type **edit** followed by a space and then the filename which you have selected, for example "text". When you have completed the command, press carriage return and wait for edit's response:

% **edit text** *(followed by a carriage return)*
"text" No such file or directory
:

If you typed the command correctly, you will now be in communication with edit. Edit has set aside a buffer for use as a temporary working space during your current editing session. It also checked to see if the file you named, "text", already existed. As we expected, it was unable to find such a file since "text" is the name of the new file that we will create. Edit confirms this with the line:

```
"text" No such file or directory
```

On the next line appears edit's prompt ":", announcing that edit expects a command from you. You may now begin to create the new file.

The "not found" message

If you misspelled "edit" by typing"editor", your request would be handled as follows:

```
% editor
editor:  not found
%
```

Your mistake in calling edit "editor" was treated by ROS as a request for a program named "editor". Since there is no program named "editor", The system reported that the program was "not found". A new $ indicates that ROS is ready for another command, so you may enter the correct command.

A summary

Your exchange with ROS as you logged in and made contact with edit should look something like this:

```
:login: susan
Password:
Welcome to Ridge Computers
% edit text
"text" No such file or directory
:
```

Entering text

You may now begin to enter text into the buffer. This is done by *appending* text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing; in effect, you are creating text. Most edit commands have two forms: a word which describes what the command does and a shorter abbreviation of that word. Either form may be used. Many beginners find the full command names easier to remember, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is "append" which may be abbreviated "a". Type **append** and press carriage return.

```
% edit text
: append
```

Messages from *edit*

If you make a mistake in entering a command and type something that edit does not recognize, edit will respond with a message intended to help you diagnose your error. For example, if you misspell the command to input text by typing, perhaps, "add" instead of "append" or "a", you will receive this message:

```
: add
add: Not an editor command
:
```

When you receive a diagnostic message, check what you typed in order to determine what part of your command confused edit. The message above means that edit was unable to recognize your mistyped command and, therefore, did not execute it. Instead, a new ":" appeared to let you know that edit is again ready to execute a command.

Text input mode

By giving the command "append" (or using the abbreviation "a"), you entered *text input mode,* also known as *append mode.* When you enter text input mode, edit responds by doing nothing. You will not receive any prompts while in text input mode. This is your signal that you are to begin entering lines of text. You can enter pretty much anything you want on the lines. The lines are transmitted one by one to the buffer and held there during the editing session. You may append as much text as you want, and *when you wish to stop entering text lines you should type a period as the only character on the line and press carriage return.* When you give this signal that you want to stop appending text, you will exit from text input mode and reenter command mode. Edit will again prompt you for a command by printing ":".

Leaving append mode does not destroy the text in the buffer. You have to leave append mode to do any of the other kinds of editing, such as changing, adding, or printing text. If you type a period as the first character and type any other character on the same line, edit will believe you want to remain in append mode and will not let you out. As this can be very frustrating, be sure to type **only** the period and carriage return.

This is as good a place as any to learn an important lesson about computers and text: a blank space is a character as far as a computer is concerned. If you so much as type a period followed by a blank (that is, type a period and then the space bar on the keyboard), you will remain in append mode with the last line of text being:

```
.
```

Enter the following lines exactly, including "thiss":

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
.
```

The last line is the period followed by a carriage return that gets you out of append mode. If while typing the line you hit an incorrect key, recall that you may delete the incorrect character or cancel the entire line of input by erasing in the usual way. Erasing a character or cancelling a line must be done before the line has been completed by a carriage return. We will discuss changes in lines already typed in session 2.

Writing text to disk

You are now ready to edit the text. The simplest kind of editing is to write it to disk as a file for safekeeping after the session is over. This is the only way to save information from one session to the next, since the editor's buffer is temporary and will last only until the end of the editing session. Thus, learning how to write a file to disk is second in importance only to entering the text. To write the contents of the buffer to a disk file, use the command "write" (or its abbreviation "w"):

```
:write
```

Edit will copy the buffer to a disk file. If the file does not yet exist, a new file will be created automatically and the presence of a "[New file]" will be noted. The newly-created file will be given the name specified when you entered the editor, in this case "text". To confirm that the disk file has been successfully written, edit will repeat the filename and give the number of lines and the total number of characters in the file. The buffer remains unchanged by the "write" command. All of the lines which were written to disk will still be in the buffer, should you want to modify or add to them.

Edit must have a filename to use before it can write a file. If you forgot to indicate the name of the file when you began the editing session, edit will print

```
No current filename
```

in response to your write command. If this happens, you can specify the filename in a new write command:

```
:write text
```

After the "write" (or "w") type a space and then the name of the file.

Signing off

We have done enough for this first lesson on using the text editor, and are ready to quit the session with edit. To do this we type "quit" (or "q") and press carriage return:

```
:write
"text" [New file]  3 lines, 90 characters
:quit
%
```

The $ is from ROS to tell you that your session with edit is over and you may command ROS further. Since we want to end the entire session at the terminal we also need to exit from ROS. In response to the ROS prompt of "$" type a "control d". This is done by holding down the control key (usually labeled "CTRL") and simultaneously pressing the d key. This ends your session and prepares the terminal for the next user. It is always important to type a "control-d" at the end of a session to make absolutely sure no one could accidentally stumble into your abandoned session and thus gain access to your files, tempting even the most honest of souls.

This is the end of the first session on text editing.

Session 2

Log in as in the first session:

```
:login: susan  (carriage return)
Password:      (give password and carriage return)
$
```

Now when you enter the command to invoke the editor, you can specify the name of the file you worked on last time. This will start edit working and it will fetch the contents of the file into the buffer. so that you can resume editing the same file. When edit has copied the file into the buffer, it will repeat its name and tell you the number of lines and characters it contains. Thus,

```
%edit text
"text" 3 lines, 90 characters
:
```

means you asked edit to fetch the file named "text" for editing, causing it to copy the 90 characters of text into the buffer. Edit awaits your further instructions. In this session, we will append more text to our file, print the contents of the buffer, and learn to change the text of a line.

Adding more text to the file

If you want to add more to the end of your text you may do so by using the append command to enter text input mode. Here we'll use the abbreviation for the append command, "a":

```
:a
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
.
```

Interrupt

Should you press the RUBOUT key (sometimes labeled DELETE) while working with edit, it will send this message to you:

```
Interrupt
:
```

Any command that edit might be executing is terminated by rubout or delete, causing edit to prompt you for a new command. If you are appending text at the time, you will exit from append mode and be expected to give another command. The line of text that you were typing when the append command was interrupted will not be entered into the buffer.

Making corrections

It is possible to erase individual letters that you have typed. This is done by typing the designated erase character, usually the number sign (#), as many times as there are characters you want to erase. If you make a bad start in a line and would like to begin again, this technique is cumbersome – what if you had 15 characters in your line and wanted to get rid of them? To do so either requires:

```
This is yukky tex###############
```

with no room for the great text you'd like to type, or,

```
This is yukky tex@ This is great text.
```

When you type the at-sign (@), you erase the entire line typed so far. You may immediately begin to retype the line. This, unfortunately, does not help after you type the line and press carriage return. To make corrections in lines which have been completed, it is necessary to use the editing commands covered in this session and those that follow.

Listing what's in the buffer

Having appended text to what you wrote in Lesson 1, you might be curious to see what is in the buffer. To print the contents of the buffer, type the command:

```
:1,$p
```

The "1" stands for line 1 of the buffer, the "$" is a special symbol designating the last line of the buffer, and "p" (or **print**) is the command to print from line 1 to the end of the buffer. Thus, "1,$p" gives you:

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
```

Occasionally, you may enter into the buffer a character which can't be printed, which is done by striking a key while the CTRL key is depressed. In printing lines, edit uses a special notation to show the existence of non-printing characters. Suppose you had introduced the non-printing character "control-a" into the word "illustrate" by accidently holding down the CTRL key while typing "a". Edit would display

```
it does illustr^Ate the editor.
```

if you asked to have the line printed. To represent the control-a, edit shows "^A". The sequence "^" followed by a capital letter stands for the one character entered by holding down the CTRL key and typing the letter which appears after the "^". We'll soon discuss the commands which can be used to correct this typing error.

In looking over the text we see that "this" is typed as "thiss" in the second line, as suggested. Let's correct the spelling.

Finding things in the buffer

In order to change something in the buffer we first need to find it. We can find "thiss" in the text we have entered by looking at a listing of the lines. Physically speaking, we search the lines of text looking for "thiss" and stop searching when we have found it. The way to tell edit to search for something is to type it inside slash marks:

```
:/thiss/
```

By typing **/thiss/** and pressing carriage return edit is instructed to search for "thiss". If we asked edit to look for a pattern of characters which it could not find in the buffer, it would respond "Pattern not found". When edit finds the characters "thiss", it will print the line of text for your inspection:

```
And thiss is some more text.
```

Edit is now positioned in the buffer at the line which it just printed, ready to make a change in the line.

The current line

At all times during an editing session, edit keeps track of the line in the buffer where it is positioned. In general, the line which has been most recently printed, entered, or changed is considered to be the current position in the buffer. You can refer to your current position in the buffer by the symbol period (.) usually known by the name "dot". If you type "." and carriage return you will be instructing edit to print the current line:

```
:.
And thiss is some more text.
```

If you want to know the number of the current line, you can type **.=** and carriage return, and edit will respond with the line number:

```
:.=
2
```

If you type the number of any line and a carriage return, edit will position you at that line and print its contents:

```
:2
And thiss is some more text.
```

You should experiment with these commands to assure yourself that you understand what they do.

Numbering lines (nu)

The **number (nu)** command is similar to print, giving both the number and the text of each printed line. To see the number and the text of the current line type

```
:nu
          2   And thiss is some more text.
```

Notice that the shortest abbreviation for the number command is "nu" (and not "n" which is used for a different command). You may specify a range of lines to be listed by the number command in the same way that lines are specified for print. For example, "1,$nu" lists all lines in the buffer with the corresponding line numbers.

Substitute command (s)

Now that we have found our misspelled word it is time to change it from "thiss" to "this". As far as edit is concerned, changing things is a matter of substituting one thing for another. As *a* stood for *append*, so *s* stands for *substitute*. We will use the abbreviation "s" to reduce the chance of mistyping the substitute command. This command will instruct edit to make the change:

```
2s/thiss/this/
```

We first indicate the line to be changed, line 2, and then type an "s" to indicate we want substitution. Inside the first set of slashes are the characters that we want to change, followed by the characters to replace them and then a closing slash mark. To summarize:

2s/ *what is to be changed* / *what to change to* /

If edit finds an exact match of the characters to be changed it will make the change **only** in the first occurrence of the characters. If it does not find the characters to be changed it will respond:

```
Substitute pattern match failed
```

indicating your instructions could not be carried out. When edit does find the characters which you want to change, it will make the substitution and automatically print the changed line, so that you can check that the correct substitution was made. In the example,

```
:2s/thiss/this/
And this is some more text.
:
```

line 2 (and line 2 only) will be searched for the characters "thiss", and when the first exact match is found, "thiss" will be changed to "this". Strictly speaking, it was not necessary above to specify the number of the line to be changed. In

```
:s/thiss/this/
```

edit will assume that we mean to change the line where we are currently positioned ("."). In this case, the command without a line number would have produced the same result because we were already positioned at the line we wished to change.

For another illustration of substitution we may choose the line:

```
Text editing is strange, but nice.
```

We might like to be a bit more positive. Thus, we could take out the characters "strange, but " so the line would read:

```
Text editing is nice.
```

A command which will first position edit at that line and then make the substitution is:

```
:/strange/s/strange, but //
```

What we have done here is combine our search with our substitution. Such combinations are perfectly legal. This illustrates that we do not necessarily have to use line numbers to identify a line to edit. Instead, we may identify the line we want to change by asking edit to search for a specified pattern of letters which occurs in that line. The parts of the above command are:

/strange/	tells edit to find the characters "strange" in the text
s	tells edit we want to make a substitution
/strange, but //	substitutes nothing at all for the characters "strange, but "

You should note the space after "but" in "/strange, but /". If you do not indicate the space is to be taken out, your line will be:

```
Text editing is  nice.
```

which looks a little funny because of the extra space between "is" and "nice". Again, we realize from this that a blank space is a real character to a computer, and in editing text we need to be aware of spaces within a line just as we would be aware of an "a" or a "4".

Another way to list what's in the buffer (z)

Although the print command is useful for looking at specific lines in the buffer, other commands can be more convenient for viewing large sections of text. You can ask to see a screen full of text at a time by using the command **z.** If you type

```
:1z
```

edit will start with line 1 and continue printing lines, stopping either when the screen of your terminal is full or when the last line in the buffer has been printed. If you want to read the next segment of text, give the command

```
:z
```

If no starting line number is given for the z command, printing will start at the "current" line, in this case the last line printed. Viewing lines in the buffer one screen full at a time is known as paging. Paging can also be used to print a section of text on a hard-copy terminal.

Saving the modified text

This seems to be a good place to pause in our work, and so we should end the second session. If you (in haste) type "q" to quit the session your dialogue with edit will be:

```
:q
No write since last change (q! quits)
:
```

This is edit's warning that you have not written the modified contents of the buffer to disk. You run the risk of losing the work you have done during the editing session since the latest write command. Since in this lesson we have not written to disk at all, everything we have done would be lost. If we did not want to save the work done during this editing session, we would have to type "q!" to confirm that we indeed wanted to end the session immediately, losing the contents of the buffer. But because we want to preserve our work, we must type:

```
:w
"text" 6 lines, 171 characters
```

and then,

```
:q
%{control d}
```

and hang up the phone or turn off the terminal when ROS asks for a name. This is the end of the second session on text editing.

Session 3

Bringing text into the buffer (e)

Log in to ROS and make contact with edit. You should try to login without looking at the notes, but if you must then by all means do.

Did you remember to give the name of the file you wanted to edit? That is, did you enter

```
% edit text
```

or simply

```
% edit
```

Both ways get you in contact with edit, but the first way will bring a copy of the file named "text" into the buffer. If you did forget to tell edit the name of your file, you can get it into the buffer by entering:

```
:e text
"text" 6 lines, 171 characters
```

The command **edit,** which may be abbreviated "**e**", tells edit that you want to erase anything that might already be in the buffer and bring a copy of the file "text" into the buffer for editing. You may also use the edit (e) command to change files in the middle of an editing session or to give edit the name of a new file that you want to create. Because the edit command clears the buffer, you will receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disk before editing the next file.

Moving text in the buffer (m)

Edit allows you to move lines of text from one location in the buffer to another by means of the **move** (**m**) command:

```
:2,4m$
```

This command directs edit to move lines 2, 3, and 4 to the end of the buffer ($). The format for the move command is that you specify the first line to be moved, the last line to be moved, the move command "m", and the line after which the moved text is to be placed. Thus,

```
:1,6m20
```

would instruct edit to move lines 1 through 6 (inclusive) to a position after line 20 in the buffer. To move just line 4 to a position in the buffer after line 6, the command would be "4m6".

Let's move some text using the command:

```
:5,$m1
2 lines moved
it does illustrate the editor.
```

After executing a command which changes more than one line of the buffer, edit tells how many lines were affected by the change. The last moved line is printed for your inspection. If you want to see more than just the last line, use the print (p), z, or number (nu) command to view more text. The buffer should now contain:

```
This is some sample text.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
```

We can restore the original order by typing:

```
:4,$m1
```

or, combining context searching and the move command:

```
:/And this is some/,/This is text/m/This is some sample/
```

The problem with combining context searching with the move command is that the chance of making a typing error in such a long command is greater than if one types line numbers.

Copying lines (copy)

The **copy** command is used to make a second copy of specified lines, leaving the original lines where they were. Copy has the same format as the move command, for example:

```
:12,15copy $
```

makes a copy of lines 12 through 15, placing the added lines after the buffer's end ($). Experiment with the copy command so that you can become familiar with how it works. Note that the shortest abbreviation for copy is "co" (and **not** the letter "c" which has another meaning).

Deleting lines (d)

Suppose you want to delete the line

```
This is text added in Session 2.
```

from the buffer. If you know the number of the line to be deleted, you can type that number followed by "**delete**" or "**d**". This example deletes line 4:

```
:4d
It doesn't mean much here, but
```

Here "4" is the number of the line to be deleted and "delete" or "d" is the command to delete the line. After executing the delete command, edit prints the line which has become the current line (".").

If you do not happen to know the line number you can search for the line and then delete it using this sequence of commands:

```
:/added in Session 2./
This is text added in Session 2.
:d
It doesn't mean much here, but
```

The "/added in Session 2./" asks edit to locate and print the next line which contains the indicated text. Once you are sure that you have correctly specified the line that you want to delete, you can enter the delete (d) command. In this case it is not necessary to specify a line number before the "d". If no line number is given, edit deletes the current line ("."), that is, the line found by our search. After the deletion, your buffer should contain:

```
This is some sample text.
And this is some more text.
Text editing is nice.
It doesn't mean much here, but
it does illustrate the editor.
```

To delete both lines 2 and 3:

```
And this is some more text.
Text editing is nice.
```

you type

```
:2,3d
```

which specifies the range of lines from 2 to 3, and the operation on those lines – "d" for delete.

Again, this presumes that you know the line numbers for the lines to be deleted. If you do not you might combine the search command with the delete command as so:

```
:/And this is some/,/Text editing is nice./d
```

A word or two of caution:

In using the search function to locate lines to be deleted you should be **absolutely sure** the characters you give as the basis for the search will take edit to the line you want deleted. Edit will search for the first occurrence of the characters starting from where you last edited – that is, from the line you see printed if you type dot (.).

A search based on too few characters may result in the wrong lines being deleted, which edit will do as easily as if you had meant it. For this reason, it is usually safer to specify the search and then delete in two separate steps, at least until you become familiar enough with using the editor that you understand how best to specify searches. For a beginner it is not a bad idea to double-check each command before pressing carriage return to send the command on its way.

Undo (u) to the rescue

The **undo (u)** command has the ability to reverse the effects of the last command. To undo the previous command, type "u" or "undo". Undo can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give. It is possible to undo only commands which have the power to change the buffer, for example delete, append, move, copy, substitute, and even undo itself. The commands write (w) and edit (e) which interact with disk files cannot be undone, nor can commands such as print which do not change the buffer. Most importantly, the **only** command which can be reversed by undo is the last "undo-able" command which you gave.

To illustrate, let's issue an undo command. Recall that the last buffer-changing command we gave deleted the lines which were formerly numbered 2 and 3. Executing undo at this moment will reverse the effects of the deletion, causing those two lines to be replaced in the buffer.

```
:u
2 more lines in file after undo
And this is some more text.
```

Here again, edit informs you if the command affects more than one line, and prints the text of the line which is now "dot" (the current line).

More about the dot (.) and buffer end ($)

The function assumed by the symbol dot depends on its context. It can be used:

1. to exit from append mode we type dot (and only a dot) on a line and press carriage return;

2. to refer to the line we are at in the buffer.

Dot can also be combined with the equal sign to get the number of the line currently being edited:

```
:.=
```

Thus if we type ".=" we are asking for the number of the line and if we type "." we are asking for the text of the line.

In this editing session and the last, we used the dollar sign to indicate the end of the buffer in commands such as print, copy, and move. The dollar sign as a command asks edit to print the last line in the buffer. If the dollar sign is combined with the equal sign ($=) edit will print the line number corresponding to the last line in the buffer.

"." and "$" therefore represent line numbers. Whenever appropriate, these symbols can be used in place of line numbers in commands. For example

```
:.,$d
```

instructs edit to delete all lines from the current line (.) to the end of the buffer.

Moving around in the buffer (+ and –)

It is convenient during an editing session to go back and re-read a previous line. To go back, we could search for some of the text if we remember it, or enter

```
– 3p
```

This tells edit to move back to a position 3 lines before the current line (.) and print that line. We can move forward in the buffer similarly:

```
+ 2p
```

instructs edit to print the line which is 2 ahead of our current position.

You may use "+ " and "– " in any command where edit accepts line numbers. Line numbers specified with "+ " or "– " can be combined to print a range of lines. The command

```
:– 1,+ 2copy$
```

makes a copy of 4 lines: the current line, the line before it, and the two after it. The copied lines will be placed after the last line in the buffer ($).

Try typing only "– "; you will move back one line just as if you had typed "– 1p". Typing the command "+ " works similarly. You might also try typing a few plus or minus signs in a row (such as "+ + + ") to see edit's response. Typing a carriage return alone on a line is the equivalent of typing "+ 1p"; it will move you one line ahead in the buffer and print that line.

If you are at the last line of the buffer and try to move further ahead, perhaps by typing a "+ " or a carriage return alone on the line, edit will remind you that you are at the end of the buffer:

```
At end-of-file
```

Similarly, if you try to move to a position before the first line, edit will print one of these messages:

```
Nonzero address required on this command
Negative address – first buffer line is 1
```

The number associated with a buffer line is the line's "address", in that it can be used to locate the line.

Changing lines (c)

There may be occasions when you want to delete certain lines and insert new text in their place. This can be accomplished easily with the **change (c)** command. The change command instructs edit to delete specified lines and then switch to text input mode in order to accept the text which will replace them. Suppose we want to change the first two lines in the buffer:

```
This is some sample text.
And this is some more text.
```

to read

```
This text was created with the text editor.
```

To do so, you can type:

```
:1,2c
2 lines changed
This text was created with the text editor.
.
:
```

In the command **1,2c** we specify that we want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the print command. These lines will be deleted. After a carriage return enters the change command, edit notifies you if more than one line will be changed and places you in text input mode. Any text typed on the following lines will be inserted into the position where lines were deleted by the change command. You will remain in text input mode until you exit in the usual way, by typing a period alone on a line. Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

This is the end of the third session on text editing.

Session 4

This lesson covers several topics, starting with commands which apply throughout the buffer, characters with special meanings, and how to issue ROS commands while in the editor. The next topics deal with files: more on reading and writing, and methods of recovering files lost in a crash. The final section suggests sources of further information.

Making commands global (g)

One disadvantage to the commands we have used for searching or substituting is that if you have a number of instances of a word to change it appears that you have to type the command repeatedly, once for each time the change needs to be made. Edit, however, provides a way to make commands apply to the entire contents of the buffer – the **global (g)** command.

To print all lines containing a certain sequence of characters, like the characters "text", the command is:

```
:g/text/p
```

The "g" instructs edit to make a global search for all lines in the buffer containing the characters "text". The "p" prints the lines found.

To issue a global command, start by typing a "g" and then a search pattern identifying the lines to be affected. Then, on the same line, type the command to be executed on the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word "text" to the word "material" the command would be a combination of the global search and the substitute command:

```
:g/text/s/text/material/g
```

Note the "g" at the end of the global command which instructs edit to change each and every instance of "text" to "material". If you do not type the "g" at the end of the command only the *first* instance of "text" in each line will be changed (the normal result of the substitute command). The "g" at the end of the command is independent of the "g" at the beginning. You may give a command such as:

```
:14s/text/material/g
```

to change every instance of "text" in line 14 alone. Further, neither command will change "Text" to "material" because "Text" begins with a capital rather than a lower-case *t*.

Edit does not automatically print the lines modified by a global command. If you want the lines to be printed, type a "p" at the end of the global command:

```
:g/text/s/text/material/gp
```

The usual qualification should be made about using the global command in combination with any other – in essence, be sure of what you are telling edit to do to the entire buffer. For example,

```
:g/ /d
72 less lines in file after global
```

will delete every line containing a blank anywhere in it. This could adversely affect your document, since most lines have spaces between words and thus would be deleted. After executing the global command, edit will print a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. You should experiment with the global command on a small buffer of text to see what it can do for you.

More about searching and substituting

In using slashes to identify a character string that we want to search for or change, we have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change "noun" to "nouns" we may type either

```
:/noun/s/noun/nouns/
```

as we have done in the past, or a somewhat abbreviated command:

```
:/noun/s//nouns/
```

In this example, the characters to be changed are not specified – there are no characters, not even a space, between the two slash marks which indicate what is to be changed. This lack of characters between the slashes is taken by the editor to mean "use the characters we last searched for as the characters to be changed."

Similarly, the last context search may be repeated by typing a pair of slashes with nothing between them:

```
:/does/
It doesn't mean much here, but
://
it does illustrate the editor.
```

Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters "does".

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the character string.

Special characters

Two characters have special meanings when used in specifying searches: "$" and "^". "$" is taken by the editor to mean "end of the line" and is used to identify strings which occur at the end of a line.

```
:g/ing$/s//ed/p
```

tells the editor to search for all lines ending in "ing" (and nothing else, not even a blank space), to change each final "ing" to "ed" and print the changed lines.

The symbol "^" indicates the beginning of a line. Thus,

```
:s/^/1. /
```

instructs the editor to insert "1." and a space at the beginning of the current line.

The characters "$" and "^" have special meanings only in the context of searching. At other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to temporarily lose its special significance by typing another special character, the backslash (\), before it.

```
:s/\$/dollar/
```

looks for the character "$" in the current line and replaces it by the word "dollar". Were it not for the backslash, the "$" would have represented "the end of the line" in your search, not necessarily the character "$". The backslash retains its special significance at all times.

Issuing ROS commands from the editor

After creating several files with the editor, you may want to delete files no longer useful to you or ask for a list of your files. Removing and listing files are not functions of the editor, and so they require the use of ROS commands. You can execute ROS commands without exiting the editor if you type a the system escape character (!) before the command name. To use the ROS *rm* command to remove the file named "junk" type:

```
:!rm junk
!
:
```

The exclamation mark (!) indicates that the rest of the line is to be processed as a ROS command. If the buffer contents have not been written since the last change, a warning will be printed before the command is executed. The editor prints a "!" when the command is completed.

Filenames and file manipulation

Throughout each editing session, edit keeps track of the name of the file being edited as the *current filename*. Edit remembers as the current filename the name given when you entered the editor. The current filename changes whenever the edit (e) command is used to specify a new file. Once edit has recorded a current filename, it inserts that name into any command where a filename has been omitted. If a write command does not specify a file, edit, as we have seen, supplies the current filename. You can have the editor write onto a different file by including its name in the write command:

```
:w chapter3
"chapter3" 283 lines, 8698 characters
```

The current filename remembered by the editor *will not be changed as a result of the write command unless it is the first filename given in the editing session.* Thus, in the next write command which does not specify a name, edit will write onto the current file and not onto the file "chapter3".

The file (f) command

To ask for the current filename, type **file** (or **f**). In response, the editor provides current information about the buffer, including the filename, your current position, and the number of lines in the buffer:

```
:f
"text" [Modified] line 3 of 4 --75%--
```

If the contents of the buffer have changed since the last time the file was written, the editor will tell you that the file has been "[Modified]". After you save the changes by writing onto a disk file, the buffer will no longer be considered modified:

```
:w
"text" 4 lines, 88 characters
:f
"text" line 3 of 4 --75%--
```

Reading additional files (r)

The **read (r)** command allows you to add the contents of a file to the buffer without destroying the text already there. To use it, specify the line after which the new text will be placed, the command *r*, and then the name of the file.

```
:$r bibliography
"bibliography" 18 lines, 473 characters
```

This command reads in the file *bibliography* and adds it to the buffer after the last line. The current filename is not changed by the read command unless it is the first filename given in the editing session.

Writing parts of the buffer

The **write (w)** command can write all or part of the buffer to a file you specify. We are already familiar with writing the entire contents of the buffer to a disk file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command, for example

```
:45,$w ending
```

Here all lines from 45 through the end of the buffer are written onto the file named *ending*. The lines remain in the buffer as part of the document you are editing, and you may continue to edit the entire buffer.

Recovering files

Under most circumstances, edit's crash recovery mechanism is able to save work to within a few lines of changes after a crash or if the phone is hung up accidently. If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login which gives the name of the recovered file. To recover the file, enter the editor and type the command **recover (rec)**, followed by the name of the lost file.

```
:recover chap6
```

Recover is sometimes unable to save the entire buffer successfully, so always check the contents of the saved buffer carefully before writing it back onto the original file.

Other recovery techniques

If something goes wrong when you are using the editor, it may be possible to save your work by using the command **preserve (pre)**, which saves the buffer as if the system had crashed. If you are writing a file and you get the message ''Quota exceeded'', you have tried to use more disk storage than is allotted to your account. *Proceed with caution* because it is likely that only a part of the editor's buffer is now present in the file you tried to write. In this case, use the system escape character (!) and the rm command to remove some files you don't need, and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

```
:preserve
```

and then seek help. Do not simply leave the editor. If you do, the buffer will be lost, and you may not be able to save your file. After a preserve, you can use the recover command once the problem has been corrected.

If you make an undesirable change to the buffer and issue a write command before discovering your mistake, the modified version will replace any previous version of the file. Should you ever lose a good version of a document in this way, do not panic and leave the editor. As long as you stay in the editor, the contents of the buffer remain accessible. Depending on the nature of the problem, it may be possible to restore the buffer to a more complete state with the undo command. After fixing the damaged buffer, you can again write the file to disk.

Further reading and other information

Edit is an editor designed for beginning and casual users. It is actually a version of a more powerful editor called *ex*. These lessons are intended to introduce you to the editor and its more commonly-used commands. We have not covered all of the editor's commands, just a selection of commands which should be sufficient to accomplish most of your editing tasks.

Using *ex*

As you become more experienced with using the editor, you may still find that edit continues to meet your needs. However, should you become interested in using ex, it is easy to switch. To begin an editing session with ex, use the name **ex** in your command instead of **edit**.

Edit commands work the same way in ex, but the editing environment is somewhat different. You should be aware of a few differences that exist between the two versions of the editor. In edit, only the characters ''^'', ''$'', and ''\'' have special meanings in searching the buffer or indicating characters to be changed by a substitute command. Several additional characters have ''magic'' meanings in ex, as described in the *Ex Reference Manual.* Another feature of the edit environment prevents users from accidently entering two alternative modes of editing, ***open*** and ***visual,*** in which the editor behaves quite differently than in normal command mode. If you are using ex and the editor behaves strangely, you may have accidently entered open mode by typing ''o''. Type the ESC key and then a ''q'' to get out of open or visual mode and back into the regular editor command mode. The document *An Introduction to Display Editing with Vi* provides a full discussion of visual mode.

Index

addressing, *see* line numbers
append mode, 4
backslash (\), 17
buffer, 1
command mode, 4
context search, 7, 9, 13, 17
control characters ("^" notation), 7
control-d, 5
current filename, 18, 19
current line (.), 8, 14
diagnostic messages, 3
disk, 1
documentation, 20
edit (to begin editing session), 2, 6
editing commands:
- append (a), 3, 4, 6
- change (c), 15
- copy (co), 12
- delete (d), 12-13
- edit (e), 11
- file (f), 18
- global (g), 16-17
- move (m), 11-12
- number (nu), 8
- preserve (pre), 19
- print (p), 7
- quit (q), 5, 10
- quit! (q!), 10
- read (r), 18-19
- recover (rec), 19
- substitute (s), 8-9, 16, 17
- undo (u), 13, 17
- write (w), 4-5, 10, 19
- z, 9
- ! (system escape), 18
- $= , 14
- + , 14
- − , 14
- //, 7, 17
- ??, 17
- ., 8, 14
- .=, 8, 14

erasing
- characters (#), 6
- lines (@), 7

ex (text editor), 20
Ex Reference Manual, 20
file, 1
file recovery, 19
filename, 1
Interrupt (message), 6
line numbers, *see also* current line
- dollar sign ($), 7, 14
- dot (.), 8, 14
- relative (+ and −), 14

logging out, 5
login procedure, 2
"magic" characters, 20
non-printing characters, 7
"not found" (message), 3
program, 1
recovery *see* file recovery
system escape (!), 18
special characters (^, $, \), 17
text input mode, 4

Ex/Edit Command Summary (Version 2.0)

Ex and *edit* are text editors, used for creating and modifying files of text on the UNIX computer system. *Edit* is a variant of *ex* with features designed to make it less complicated to learn and use. In terms of command syntax and effect the editors are essentially identical, and this command summary applies to both.

The summary is meant as a quick reference for users already acquainted with *edit* or *ex*. Fuller explanations of the editors are available in the documents *Edit: A Tutorial* (a self-teaching introduction) and the *Ex Reference Manual* (the comprehensive reference source for both *edit* and *ex*).

In the examples included with the summary, commands and text entered by the user are printed in **boldface** to distinguish them from responses printed by the computer.

The Editor Buffer

In order to perform its tasks the editor sets aside a temporary work space, called a *buffer*, separate from the user's permanent file. Before starting to work on an existing file the editor makes a copy of it in the buffer, leaving the original untouched. All editing changes are made to the buffer copy, which must then be written back to the permanent file in order to update the old version. The buffer disappears at the end of the editing session.

Editing: Command and Text Input Modes

During an editing session there are two usual modes of operation: *command* mode and *text input* mode. (This disregards, for the moment, *open* and *visual* modes, discussed below.) In command mode, the editor issues a colon prompt (:) to show that it is ready to accept and execute a command. In text input mode, on the other hand, there is no prompt and the editor merely accepts text to be added to the buffer. Text input mode is initiated by the commands *append, insert,* and *change,* and is terminated by typing a period as the first and only character on a line.

Line Numbers and Command Syntax

The editor keeps track of lines of text in the buffer by numbering them consecutively starting with 1 and renumbering as lines are added or deleted. At any given time the editor is positioned at one of these lines; this position is called the *current line*. Generally, commands that change the contents of the buffer print the new current line at the end of their execution.

Most commands can be preceded by one or two line-number addresses which indicate the lines to be affected. If one number is given the command operates on that line only; if two, on an inclusive range of lines. Commands that can take line-number prefixes also assume default prefixes if none are given. The default assumed by each command is designed to make it convenient to use in many instances without any line-number prefix. For the most part, a command used without a prefix operates on the current line, though exceptions to this rule should be noted. The *print* command by itself, for instance, causes one line, the current line, to be printed at the terminal.

The summary shows the number of line addresses that can be prefixed to each command as well as the defaults assumed if they are omitted. For example, *(.,.)* means that up to 2 line-numbers may be given, and that if none is given the command operates on the current line. (In the address prefix notation, "." stands for the current line and "$" stands for the last line of the buffer.) If no such notation appears, no line-number prefix may be used.

Some commands take trailing information; only the more important instances of this are mentioned in the summary.

Open and Visual Modes

Besides command and text input modes, *ex* and *edit* provide on some CRT terminals other modes of editing, *open* and *visual*. In these modes the cursor can be moved to individual words or characters in a line. The commands then given are very different from the standard editor commands; most do not appear on the screen when typed. *An Introduction to Display Editing with Vi* provides a full discussion.

Special Characters

Some characters take on special meanings when used in context searches and in patterns given to the *substitute* command. For *edit*, these are "^" and "$", meaning the beginning and end of a line, respectively. *Ex* has the following additional special characters:

. & * [ ] ~

To use one of the special characters as its simple graphic representation rather than with its special meaning, precede it by a backslash (\). The backslash always has a special meaning.

Name	Abbr	Description	Examples
(.)**append**	**a**	Begins text input mode, adding lines to the buffer after the line specified. Appending continues until "." is typed alone at the beginning of a new line, followed by a carriage return. *0a* places lines at the beginning of the buffer.	:**a** **Three lines of text** **are added to the buffer** **after the current line.** **.** :
(.,.)**change**	**c**	Deletes indicated line(s) and initiates text input mode to replace them with new text which follows. New text is terminated the same way as with *append.*	:**5,6c** **Lines 5 and 6 are** **deleted and replaced by** **these three lines.** **.** :
(.,.)**copy** *addr*	**co**	Places a copy of the specified lines after the line indicated by *addr.* The example places a copy of lines 8 through 12, inclusive, after line 25.	:**8,12co 25** Last line copied is printed :
(.,.)**delete**	**d**	Removes lines from the buffer and prints the current line after the deletion.	:**13,15d** New current line is printed :
edit *file* **edit!** *file*	**e** **e!**	Clears the editor buffer and then copies into it the named *file,* which becomes the current file. This is a way of shifting to a different file without leaving the editor. The editor issues a warning message if this command is used before saving changes made to the file already in the buffer; using the form **e!** overrides this protective mechanism.	:**e ch10** No write since last change :**e! ch10** "ch10" 3 lines, 62 characters :
file *name*	**f**	If followed by a *name,* renames the current file to *name.* If used without *name,* prints the name of the current file.	:**f ch9** "ch9" [Modified] 3 lines ... :**f** "ch9" [Modified] 3 lines ... :
(1,$)**global** (1,$)**global!**	**g** **g!** or **v**	**global**/*pattern*/*commands* Searches the entire buffer (unless a smaller range is specified by line-number prefixes) and executes *commands* on every line with an expression matching *pattern.* The second form, abbreviated either **g!** or **v**, executes *commands* on lines that *do not* contain the expression *pattern.*	:**g/nonsense/d** :
(.)**insert**	**i**	Inserts new lines of text immediately before the specified line. Differs from *append* only in that text is placed before, rather than after, the indicated line. In other words, **1i** has the same effect as **0a**.	:**1i** **These lines of text will** **be added prior to line 1.** **.** :
(.,.+1)**join**	**j**	Join lines together, adjusting white space (spaces and tabs) as necessary.	:**2,5j** Resulting line is printed :

Name	Abbr	Description	Examples
(.,.)**list**	**l**	Prints lines in a more unambiguous way than the *print* command does. The end of a line, for example, is marked with a "$", and tabs printed as "^I".	:**9l** This is line 9$:
(.,.)**move** *addr*	**m**	Moves the specified lines to a position after the line indicated by *addr*.	:**12,15m 25** New current line is printed :
(.,.)**number**	**nu**	Prints each line preceded by its buffer line number.	:**nu** 10 This is line 10 :
(.)**open**	**o**	Too involved to discuss here, but if you enter open mode accidentally, press the ESC key followed by **q** to get back into normal editor command mode. *Edit* is designed to prevent accidental use of the open command.	
preserve	**pre**	Saves a copy of the current buffer contents as though the system had just crashed. This is for use in an emergency when a *write* command has failed and you don't know how else to save your work.†	:**preserve** File preserved. :
(.,.)**print**	**p**	Prints the text of line(s).	:**+2,+3p** The second and third lines after the current line :
quit **quit!**	**q** **q!**	Ends the editing session. You will receive a warning if you have changed the buffer since last writing its contents to the file. In this event you must either type **w** to write, or type **q!** to exit from the editor without saving your changes.	:**q** No write since last change :**q!** %
(.)**read** *file*	**r**	Places a copy of *file* in the buffer after the specified line. Address 0 is permissible and causes the copy of *file* to be placed at the beginning of the buffer. The *read* command does not erase any text already in the buffer. If no line number is specified, *file* is placed after the current line.	:**0r newfile** "newfile" 5 lines, 86 characters :
recover *file*	**rec**	Retrieves a copy of the editor buffer after a system crash, editor crash, phone line disconnection, or *preserve* command.	
(.,.)**substitute**	**s**	**substitute/***pattern***/***replacement***/** **substitute/***pattern***/***replacement***/gc** Replaces the first occurrence of *pattern* on a line with *replacement*. Including a **g** after the command changes all occurrences of *pattern* on the line. The **c** option allows the user to confirm each substitution before it is made; see the manual for details.	:**3p** Line 3 contains a misstake :**s/misstake/mistake/** Line 3 contains a mistake :

† Seek assistance from a consultant as soon as possible after saving a file with the *preserve* command, because the file is saved on system storage space for only one week.

Name	Abbr	Description	Examples
undo	**u**	Reverses the changes made in the buffer by the last buffer-editing command. Note that this example contains a notification about the number of lines affected.	:**1,15d** 15 lines deleted new line number 1 is printed :**u** 15 more lines in file ... old line number 1 is printed :
(1,$)**write** *file* (1,$)**write!** *file*	**w** **w!**	Copies data from the buffer onto a permanent file. If no *file* is named, the current filename is used. The file is automatically created if it does not yet exist. A response containing the number of lines and characters in the file indicates that the write has been completed successfully. The editor's built-in protections against overwriting existing files will in some circumstances inhibit a write. The form **w!** forces the write, confirming that an existing file is to be overwritten.	:**w** "file7" 64 lines, 1122 characters :**w file8** "file8" File exists ... :**w! file8** "file8" 64 lines, 1122 characters :
(.)**z** *count*	**z**	Prints a screen full of text starting with the line indicated; or, if *count* is specified, prints that number of lines. Variants of the *z* command are described in the manual.	
!*command*		Executes the remainder of the line after **!** as a UNIX command. The buffer is unchanged by this, and control is returned to the editor when the execution of *command* is complete.	:**!date** Fri Jun 9 12:15:11 PDT 1978 ! :
control-d		Prints the next *scroll* of text, normally half of a screen. See the manual for details of the *scroll* option.	
(.+1)<cr>		An address alone followed by a carriage return causes the line to be printed. A carriage return by itself prints the line following the current line.	:<cr> the line after the current line :
/*pattern*/		Searches for the next line in which *pattern* occurs and prints it.	:**/This pattern/** This pattern next occurs here. :
//		Repeats the most recent search.	:**//** This pattern also occurs here. :
?*pattern*?		Searches in the reverse direction for *pattern*.	
??		Repeats the most recent search, moving in the reverse direction through the buffer.	

An Introduction to Display Editing with Vi

This document is based on a paper by William Joy and Mark Horton of the University of California, Berkeley.

PREFACE:: *Vi* (which stands for "visual" and is pronounced "vee-eye") is a display-oriented, interactive text editor. When using *vi*, the screen of your terminal acts as a window into the file which you are editing. Changes which you make to the file are reflected in what you see.

Using *vi* you can insert new text any place in the file quite easily. Most of the commands to *vi* move the cursor around in the file. There are commands to move the cursor forward and backward in units of characters, words, sentences and paragraphs. A small set of operators are combined with the motion commands to form operations such as delete word or change paragraph, in a simple and natural way.

The full command set of the more traditional, line oriented editor *ex* is available within *vi*; it is quite simple to switch between the two modes of editing.

1. Getting started

This document provides a quick introduction to *vi*. You should be running *vi* on a file you are familiar with while you are reading this. The first part of this document (sections 1 through 5) describes the basics of using *vi*. Some topics of special interest are presented in section 6, and some nitty-gritty details of how the editor functions are saved for section 7 to avoid cluttering the presentation here.

There is also a short appendix here, which gives for each character the special meanings which this character has in *vi*. Attached to this document should be a quick reference card. This card summarizes the commands of *vi* in a very compact format. You should have the card handy while you are learning *vi*.

1.1. Specifying terminal type

Before you can start *vi* you must tell the system what kind of terminal you are using. Here is a (necessarily incomplete) list of terminal type codes. If your terminal does not appear here, you should consult with one of the staff members on your system to find out the code for your terminal. If your terminal does not have a code, one can be assigned and a description for the terminal can be created.

Code	Full name	Type
2621	Hewlett-Packard 2621A/P	Intelligent
2645	Hewlett-Packard 264x	Intelligent
act4	Microterm ACT-IV	Dumb
act5	Microterm ACT-V	Dumb
adm3a	Lear Siegler ADM-3a	Dumb
adm31	Lear Siegler ADM-31	Intelligent
c100	Human Design Concept 100	Intelligent
dm1520	Datamedia 1520	Dumb
dm2500	Datamedia 2500	Intelligent
dm3025	Datamedia 3025	Intelligent
fox	Perkin-Elmer Fox	Dumb
h1500	Hazeltine 1500	Intelligent
h19	Heathkit h19	Intelligent

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

i100	Infoton 100	Intelligent
mime	Imitating a smart act4	Intelligent
t1061	Teleray 1061	Intelligent
vt52	Dec VT-52	Dumb

Suppose for example that you have a Hewlett-Packard HP2621A terminal. The code used by the system for this terminal is '2621'. In this case you can use one of the following commands to tell the system the type of your terminal:

```
% setenv TERM 2621
```

This command works with the shell *csh* on both version 6 and 7 systems. If you are using the standard version 7 shell then you should give the commands

```
$ TERM=2621
$ export TERM
```

If you want to arrange to have your terminal type set up automatically when you log in, you can use the *tset* program. If you dial in on a *mime*, but often use hardwired ports, a typical line for your *.login* file (if you use csh) would be

```
setenv TERM `tset - - d mime`
```

or for your *.profile* file (if you use sh)

```
TERM=`tset - - d mime`
```

Tset knows which terminals are hardwired to each port and needs only to be told that when you dial in you are probably on a *mime*. *Tset* is usually used to change the erase and kill characters, too.

1.2. Editing a file

After telling the system which kind of terminal you have, you should make a copy of a file you are familiar with, and run *vi* on this file, giving the command

```
% vi name
```

replacing *name* with the name of the copy file you just created. The screen should clear and the text of your file should appear on the screen. If something else happens refer to the footnote.‡

1.3. The editor's copy: the buffer

The editor does not directly modify the file which you are editing. Rather, the editor makes a copy of this file, in a place called the *buffer*, and remembers the file's name. You do not affect the contents of the file unless and until you write the changes you make back into the original file.

‡ If you gave the system an incorrect terminal type code then the editor may have just made a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. In this case hit the keys **:q** (colon and the q key) and then hit the RETURN key. This should get you back to the command level interpreter. Figure out what you did wrong (ask someone else if necessary) and try again.

Another thing which can go wrong is that you typed the wrong file name and the editor just printed an error diagnostic. In this case you should follow the above procedure for getting out of the editor, and try again this time spelling the file name correctly.

If the editor doesn't seem to respond to the commands which you type here, try sending an interrupt to it by hitting the DEL or RUB key on your terminal, and then hitting the **:q** command again followed by a carriage return.

1.4. Notational conventions

In our examples, input which must be typed as is will be presented in **bold face**. Text which should be replaced with appropriate input will be given in *italics*. We will represent special characters in SMALL CAPITALS.

1.5. Arrow keys

The editor command set is independent of the terminal you are using. On most terminals with cursor positioning keys, these keys will also work within the editor. If you don't have cursor positioning keys, or even if you do, you can use the **h j k** and **l** keys as cursor positioning keys (these are labelled with arrows on an *adm3a).**

(Particular note for the HP2621: on this terminal the function keys must be *shifted* (ick) to send to the machine, otherwise they only act locally. Unshifted use will leave the cursor positioned incorrectly.)

1.6. Special characters: ESC, CR and DEL

Several of these special characters are very important, so be sure to find them right now. Look on your keyboard for a key labelled ESC or ALT. It should be near the upper left corner of your terminal. Try hitting this key a few times. The editor will ring the bell to indicate that it is in a quiescent state.‡ Partially formed commands are cancelled by ESC, and when you insert text in the file you end the text insertion with ESC. This key is a fairly harmless one to hit, so you can just hit it if you don't know what is going on until the editor rings the bell.

The CR or RETURN key is important because it is used to terminate certain commands. It is usually at the right side of the keyboard, and is the same command used at the end of each shell command.

Another very useful key is the DEL or RUB key, which generates an interrupt, telling the editor to stop what it is doing. It is a forceful way of making the editor listen to you, or to return it to the quiescent state if you don't know or don't like what is going on. Try hitting the '/' key on your terminal. This key is used when you want to specify a string to be searched for. The cursor should now be positioned at the bottom line of the terminal after a '/' printed as a prompt. You can get the cursor back to the current position by hitting the DEL or RUB key; try this now.* From now on we will simply refer to hitting the DEL or RUB key as "sending an interrupt."**

The editor often echoes your commands on the last line of the terminal. If the cursor is on the first position of this last line, then the editor is performing a computation, such as computing a new position in the file after a search or running a command to reformat part of the buffer. When this is happening you can stop the editor by sending an interrupt.

1.7. Getting out of the editor

After you have worked with this introduction for a while, and you wish to do something else, you can give the command **ZZ** to the editor. This will write the contents of the editor's buffer back into the file you are editing, if you made any changes, and then quit from the editor. You can also end an editor session by giving the command **:q!**CR;† this is a dangerous but occasionally essential command which ends the editor session and discards all your changes. You need to know about this command in case you change the editor's copy of a file you wish

* As we will see later, *h* moves back to the left (like control-h which is a backspace), *j* moves down (in the same column), *k* moves up (in the same column), and *l* moves to the right.

‡ On smart terminals where it is possible, the editor will quietly flash the screen rather than ringing the bell.

* Backspacing over the '/' will also cancel the search.

** On some systems, this interruptibility comes at a price: you cannot type ahead when the editor is computing with the cursor on the bottom line.

† All commands which read from the last display line can also be terminated with a ESC as well as an CR.

only to look at. Be very careful not to give this command when you really want to save the changes you have made.

2. Moving around in the file

2.1. Scrolling and paging

The editor has a number of commands for moving around in the file. The most useful of these is generated by hitting the control and D keys at the same time, a control-D or '^D'. We will use this two character notation for referring to these control keys from now on. You may have a key labelled '^' on your terminal. This key will be represented as '↑' in this document; '^' is exclusively used as part of the '^x' notation for control characters.‡

As you know now if you tried hitting **^D**, this command scrolls down in the file. The **D** thus stands for down. Many editor commands are mnemonic and this makes them much easier to remember. For instance the command to scroll up is **^U**. Many dumb terminals can't scroll up at all, in which case hitting **^U** clears the screen and refreshes it with a line which is farther back in the file at the top.

If you want to see more of the file below where you are, you can hit **^E** to expose one more line at the bottom of the screen, leaving the cursor where it is. ‡‡ The command **^Y** (which is hopelessly non-mnemonic, but next to **^U** on the keyboard) exposes one more line at the top of the screen.

There are other ways to move around in the file; the keys **^F** and **^B** ‡ move forward and backward a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file using these rather than **^D** and **^U** if you wish.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, hitting **^F** to move forward a page will leave you only a little context to look back at. Scrolling on the other hand leaves more context, and happens more smoothly. You can continue to read the text as scrolling is taking place.

2.2. Searching, goto, and previous context

Another way to position yourself in the file is by giving the editor a string to search for. Type the character / followed by a string of characters terminated by CR. The editor will position the cursor at the next occurrence of this string. Try hitting **n** to then go to the next occurrence of this string. The character **?** will search backwards from where you are, and is otherwise like /.†

If the search string you give the editor is not present in the file the editor will print a diagnostic on the last line of the screen, and the cursor will be returned to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with an ↑. To match only at the end of a line, end the search string with a $. Thus **/↑search**CR will search for the word 'search' at the beginning of a line, and **/last$**CR searches for the word 'last' at the end of a line.*

‡ If you don't have a '^' key on your terminal then there is probably a key labelled '↑'; in any case these characters are one and the same.

‡‡ Version 3 only.

‡ Not available in all v2 editors due to memory constraints.

† These searches will normally wrap around the end of the file, and thus find the string even if it is not on a line in the direction you search provided it is anywhere else in the file. You can disable this wraparound in scans by giving the command **:se nowrapscan**CR, or more briefly **:se nows**CR.

*Actually, the string you give to search for here can be a *regular expression* in the sense of the editors *ex*(1) and *ed*(1). If you don't wish to learn about this yet, you can disable this more general facility by doing **:se nomagic**CR; by putting this command in EXINIT in your environment, you can have this always be in effect (more about *EXINIT* later.)

The command **G**, when preceded by a number will position the cursor at that line in the file. Thus **1G** will move the cursor to the first line of the file. If you give **G** no count, then it moves to the end of the file.

If you are near the end of the file, and the last line is not at the bottom of the screen, the editor will place only the character '~' on each remaining line. This indicates that the last line in the file is on the screen; that is, the '~' lines are past the end of the file.

You can find out the state of the file you are editing by typing a **^G**. The editor will show you the name of the file you are editing, the number of the current line, the number of lines in the buffer, and the percentage of the way through the buffer which you are. Try doing this now, and remember the number of the line you are on. Give a **G** command to get to the end and then another **G** command to get back where you were.

You can also get back to a previous position by using the command **``** (two back quotes). This is often more convenient than **G** because it requires no advance preparation. Try giving a **G** or a search with **/** or **?** and then a **``** to get back to where you were. If you accidentally hit **n** or any command which moves you far away from a context of interest, you can quickly get back by hitting **``**.

2.3. Moving around on the screen

Now try just moving the cursor around on the screen. If your terminal has arrow keys (4 or 5 keys with arrows going in each direction) try them and convince yourself that they work. (On certain terminals using v2 editors, they won't.) If you don't have working arrow keys, you can always use **h**, **j**, **k**, and **l**. Experienced users of *vi* prefer these keys to arrow keys, because they are usually right underneath their fingers.

Hit the **+** key. Each time you do, notice that the cursor advances to the next line in the file, at the first non-white position on the line. The **–** key is like **+** but goes the other way.

These are very common keys for moving up and down lines in the file. Notice that if you go off the bottom or top with these keys then the screen will scroll down (and up if possible) to bring a line at a time into view. The RETURN key has the same effect as the **+** key.

Vi also has commands to take you to the top, middle and bottom of the screen. **H** will take you to the top (home) line on the screen. Try preceding it with a number as in **3H**. This will take you to the third line on the screen. Many *vi* commands take preceding numbers and do interesting things with them. Try **M**, which takes you to the middle line on the screen, and **L**, which takes you to the last line on the screen. **L** also takes counts, thus **5L** will take you to the fifth line from the bottom.

2.4. Moving within a line

Now try picking a word on some line on the screen, not the first word on the line. move the cursor using RETURN and **–** to be on the line where the word is. Try hitting the **w** key. This will advance the cursor to the next word on the line. Try hitting the **b** key to back up words in the line. Also try the **e** key which advances you to the end of the current word rather than to the beginning of the next word. Also try SPACE (the space bar) which moves right one character and the BS (backspace or **^H**) key which moves left one character. The key **h** works as **^H** does and is useful if you don't have a BS key. (Also, as noted just above, **l** will move to the right.)

If the line had punctuation in it you may have noticed that that the **w** and **b** keys stopped at each group of punctuation. You can also go back and forwards words without stopping at punctuation by using **W** and **B** rather than the lower case equivalents. Think of these as bigger words. Try these on a few lines with punctuation to see how they differ from the lower case **w** and **b**.

The word keys wrap around the end of line, rather than stopping at the end. Try moving to a word on a line below where you are by repeatedly hitting **w**.

2.5. Summary

SPACE	advance the cursor one position
^B	backwards to previous page
^D	scrolls down in the file
^E	exposes another line at the bottom (v3)
^F	forward to next page
^G	tell what is going on
^H	backspace the cursor
^N	next line, same column
^P	previous line, same column
^U	scrolls up in the file
^Y	exposes another line at the top (v3)
+	next line, at the beginning
–	previous line, at the beginning
/	scan for a following string forwards
?	scan backwards
B	back a word, ignoring punctuation
G	go to specified line, last default
H	home screen line
M	middle screen line
L	last screen line
W	forward a word, ignoring punctuation
b	back a word
e	end of current word
n	scan for next instance of **/** or **?** pattern
w	word after this word

2.6. View ‡

If you want to use the editor to look at a file, rather than to make changes, invoke it as *view* instead of *vi*. This will set the *readonly* option which will prevent you from accidently overwriting the file.

3. Making simple changes

3.1. Inserting

One of the most useful commands is the **i** (insert) command. After you type **i**, everything you type until you hit ESC is inserted into the file. Try this now; position yourself to some word in the file and try inserting text before this word. If you are on an dumb terminal it will seem, for a minute, that some of the characters in your line have been overwritten, but they will reappear when you hit ESC.

Now try finding a word which can, but does not, end in an 's'. Position yourself at this word and type **e** (move to end of word), then **a** for append and then 'sESC' to terminate the textual insert. This sequence of commands can be used to easily pluralize a word.

Try inserting and appending a few times to make sure you understand how this works; **i** placing text to the left of the cursor, **a** to the right.

It is often the case that you want to add new lines to the file you are editing, before or after some specific line in the file. Find a line where this makes sense and then give the command **o** to create a new line after the line you are on, or the command **O** to create a new line before the line you are on. After you create a new line in this way, text you type up to an ESC

‡ Not available in all v2 editors due to memory constraints.

is inserted on the new line.

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lower case key and the other is given by an upper case key. In these cases, the upper case key often differs from the lower case key in its sense of direction, with the upper case key working backward and/or up, while the lower case key moves forward and/or down.

Whenever you are typing in text, you can give many lines of input or just a few characters. To type in more than one line of text, hit a RETURN at the middle of your input. A new line will be created for text, and you can continue to type. If you are on a slow and dumb terminal the editor may choose to wait to redraw the tail of the screen, and will let you type over the existing screen lines. This avoids the lengthy delay which would occur if the editor attempted to keep the tail of the screen always up to date. The tail of the screen will be fixed up, and the missing lines will reappear, when you hit ESC.

While you are inserting new text, you can use the characters you normally use at the system command level (usually **^H** or #) to backspace over the last character which you typed, and the character which you use to kill input lines (usually @, **^X**, or **^U**) to erase the input you have typed on the current line.† The character **^W** will erase a whole word and leave you after the space after the previous word; it is useful for quickly backing up in an insert.

Notice that when you backspace during an insertion the characters you backspace over are not erased; the cursor moves backwards, and the characters remain on the display. This is often useful if you are planning to type in something similar. In any case the characters disappear when when you hit ESC; if you want to get rid of them immediately, hit an ESC and then **a** again.

Notice also that you can't erase characters which you didn't insert, and that you can't backspace around the end of a line. If you need to back up to the previous line to make a correction, just hit ESC and move the cursor back to the previous line. After making the correction you can return to where you were and use the insert or append command again.

3.2. Making small corrections

You can make small corrections in existing text quite easily. Find a single character which is wrong or just pick any character. Use the arrow keys to find the character, or get near the character with the word motion keys and then either backspace (hit the BS key or **^H** or even just **h**) or SPACE (using the space bar) until the cursor is on the character which is wrong. If the character is not needed then hit the **x** key; this deletes the character from the file. It is analogous to the way you **x** out characters when you make mistakes on a typewriter (except it's not as messy).

If the character is incorrect, you can replace it with the correct character by giving the command **r***c*, where *c* is replaced by the correct character. Finally if the character which is incorrect should be replaced by more than one character, give the command **s** which substitutes a string of characters, ending with ESC, for it. If there are a small number of characters which are wrong you can precede **s** with a count of the number of characters to be replaced. Counts are also useful with **x** to specify the number of characters to be deleted.

3.3. More corrections: operators

You already know almost enough to make changes at a higher level. All you need to know now is that the **d** key acts as a delete operator. Try the command **dw** to delete a word. Try hitting **.** a few times. Notice that this repeats the effect of the **dw**. The command **.** repeats the last command which made a change. You can remember it by analogy with an ellipsis '**...**'.

† In fact, the character ^H (backspace) always works to erase the last input character here, regardless of what your erase character is.

Now try **db**. This deletes a word backwards, namely the preceding word. Try **d**SPACE. This deletes a single character, and is equivalent to the **x** command.

Another very useful operator is **c** or change. The command **cw** thus changes the text of a single word. You follow it by the replacement text ending with an ESC. Find a word which you can change to another, and try this now. Notice that the end of the text to be changed was marked with the character '$' so that you can see this as you are typing in the new material.

3.4. Operating on lines

It is often the case that you want to operate on lines. Find a line which you want to delete, and type **dd**, the **d** operator twice. This will delete the line. If you are on a dumb terminal, the editor may just erase the line on the screen, replacing it with a line with only an @ on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen which would be necessary to close up the hole created by the deletion on a terminal without a delete line capability.

Try repeating the **c** operator twice; this will change a whole line, erasing its previous contents and replacing them with text you type up to an ESC.†

You can delete or change more than one line by preceding the **dd** or **cc** with a count, i.e. **5dd** deletes 5 lines. You can also give a command like **dL** to delete all the lines up to and including the last line on the screen, or **d3L** to delete through the third from the bottom line. Try some commands like this now.* Notice that the editor lets you know when you change a large number of lines so that you can see the extent of the change. The editor will also always tell you when a change you make affects text which you cannot see.

3.5. Undoing

Now suppose that the last change which you made was incorrect; you could use the insert, delete and append commands to put the correct material back. However, since it is often the case that we regret a change or make a change incorrectly, the editor provides a **u** (undo) command to reverse the last change which you made. Try this a few times, and give it twice in a row to notice that an **u** also undoes a **u.**

The undo command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the original state of the line back. The **U** command restores the current line to the state before you started changing it.

You can recover text which you delete, even if undo will not bring it back; see the section on recovering lost text below.

3.6. Summary

SPACE	advance the cursor one position
^H	backspace the cursor
^W	erase a word during an insert
erase	your erase (usually ^H or #), erases a character during an insert
kill	your kill (usually @, ^X, or ^U), kills the insert on this line
.	repeats the changing command
O	opens and inputs new lines, above the current
U	undoes the changes you made to the current line
a	appends text after the cursor
c	changes the object you specify to the following text

† The command S is a convenient synonym for for **cc**, by analogy with s. Think of S as a substitute on lines, while s is a substitute on characters.

* One subtle point here involves using the / search after a **d**. This will normally delete characters from the current position to the point of the match. If what is desired is to delete whole lines including the two points, give the pattern as **/pat/+0**, a line address.

d	deletes the object you specify
i	inserts text before the cursor
o	opens and inputs new lines, below the current
u	undoes the last change

4. Moving about; rearranging and duplicating text

4.1. Low level character motions

Now move the cursor to a line where there is a punctuation or a bracketing character such as a parenthesis or a comma or period. Try the command **f***x* where *x* is this character. This command finds the next *x* character to the right of the cursor in the current line. Try then hitting a **;**, which finds the next instance of the same character. By using the **f** command and then a sequence of **;**'s you can often get to a particular place in a line much faster than with a sequence of word motions or SPACEs. There is also a **F** command, which is like **f**, but searches backward. The **;** command repeats **F** also.

When you are operating on the text in a line it is often desirable to deal with the characters up to, but not including, the first instance of a character. Try **df***x* for some *x* now and notice that the *x* character is deleted. Undo this with **u** and then try **dt***x*; the **t** here stands for to, i.e. delete up to the next *x*, but not the *x*. The command **T** is the reverse of **t**.

When working with the text of a single line, an **↑** moves the cursor to the first non-white position on the line, and a **$** moves it to the end of the line. Thus **$a** will append new text at the end of the current line.

Your file may have tab (**^I**) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every 8 positions.* When the cursor is at a tab, it sits on the last of the several spaces which represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may have nonprinting characters in it. These characters are displayed in the same way they are represented in this document, that is with a two character code, the first character of which is '^'. On the screen non-printing characters resemble a '^' character adjacent to another, but spacing or backspacing over the character will reveal that the two characters are, like the spaces representing a tab character, a single character.

The editor sometimes discards control characters, depending on the character and the setting of the *beautify* option, if you attempt to insert them in your file. You can get a control character in the file by beginning an insert and then typing a **^V** before the control character. The **^V** quotes the following character, causing it to be inserted directly into the file.

4.2. Higher level text objects

In working with a document it is often advantageous to work in terms of sentences, paragraphs, and sections. The operations **(** and **)** move to the beginning of the previous and next sentences respectively. Thus the command **d)** will delete the rest of the current sentence; likewise **d(** will delete the previous sentence if you are at the beginning of the current sentence, or the current sentence up to where you are if you are not at the beginning of the current sentence.

A sentence is defined to end at a '.', '!' or '?' which is followed by either the end of a line, or by two spaces. Any number of closing ')', ']', '"' and '' characters may appear after the '.', '!' or '?' before the spaces or end of line.

The operations **{** and **}** move over paragraphs and the operations **[[** and **]]** move over sections.†

* This is settable by a command of the form **:se ts=***x*CR, where *x* is 4 to set tabstops every four columns. This has effect on the screen representation within the editor.

† The [[and]] operations require the operation character to be doubled because they can move the cursor far

A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the definition of the string valued option *paragraphs.* The default setting for this option defines the paragraph macros of the – *ms* and – *mm* macro packages, i.e. the '.IP', '.LP', '.PP' and '.QP', '.P' and '.LI' macros.‡ Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs.

Sections in the editor begin after each macro in the *sections* option, normally '.NH', '.SH', '.H' and '.HU', and each line with a formfeed **^L** in the first column. Section boundaries are always line and paragraph boundaries also.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking through it using the section commands. The section commands interpret a preceding count as a different window size in which to redraw the screen at the new location, and this window size is the base size for newly drawn windows until another size is specified. This is very useful if you are on a slow terminal and are looking for a particular section. You can give the first section command a small count to then see each successive section heading in a small window.

4.3. Rearranging and duplicating text

The editor has a single unnamed buffer where the last deleted or changed away text is saved, and a set of named buffers **a**– **z** which you can use to save copies of text and to move text around in your file and between files.

The operator **y** yanks a copy of the object which follows into the unnamed buffer. If preceded by a buffer name, **"*x*y**, where *x* here is replaced by a letter **a**– **z**, it places the text in the named buffer. The text can then be put back in the file with the commands **p** and **P**; **p** puts the text after or below the cursor, while **P** puts the text before or above the cursor.

If the text which you yank forms a part of a line, or is an object such as a sentence which partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if you use **P**). If the yanked text forms whole lines, they will be put back as whole lines, without changing the current line. In this case, the put acts much like a **o** or **O** command.

Try the command **YP**. This makes a copy of the current line and leaves you on this copy, which is placed before the current line. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** will also make a copy of the current line, and place it after the current line. You can give **Y** a count of lines to yank, and thus duplicate several lines; try **3YP**.

To move text within the buffer, you need to delete it in one place, and put it back in another. You can precede a delete operation by the name of a buffer in which the text is to be stored as in **"a5dd** deleting 5 lines into the named buffer *a*. You can then move the cursor to the eventual resting place of the these lines and do a **"ap** or **"aP** to put them back. In fact, you can switch and edit another file before you put the lines back, by giving a command of the form **:e** *name*CR where *name* is the name of the other file you want to edit. You will have to write back the contents of the current editor buffer (or discard them) if you have made changes before the editor will let you switch to the other file. An ordinary delete command saves the text in the unnamed buffer, so that an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you change files, so to move text from one file to another you should use a~~n~~ ~~un~~named buffer.

from where it currently is. While it is easy to get back with the command ``, these commands would still be frustrating if they were easy to hit accidentally.

‡ You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See section 6.2 for details. The '.bp' directive is also considered to start a paragraph.

4.4. Summary.

↑	first non-white on line
$	end of line
)	forward sentence
}	forward paragraph
]]	forward section
(	backward sentence
{	backward paragraph
[[	backward section
f*x*	find *x* forward in line
p	put text back, after cursor or below current line
y	yank operator, for copies and moves
t*x*	up to *x* forward, for operators
F*x*	f backward in line
P	put text back, before cursor or above current line
T*x*	t backward in line

5. High level commands

5.1. Writing, quitting, editing new files

So far we have seen how to enter *vi* and to write out our file using either **ZZ** or **:w**CR. The first exits from the editor, (writing if changes were made), the second writes and stays in the editor.

If you have changed the editor's copy of the file but do not wish to save your changes, either because you messed up the file or decided that the changes are not an improvement to the file, then you can give the command **:q!**CR to quit from the editor without writing the changes. You can also reedit the same file (starting over) by giving the command **:e!**CR. These commands should be used only rarely, and with caution, as it is not possible to recover the changes you have made after you discard them in this manner.

You can edit a different file without leaving the editor by giving the command **:e** *name*CR. If you have not written out your file before you try to do this, then the editor will tell you this, and delay editing the other file. You can then give the command **:w**CR to save your work and then the **:e** *name*CR command again, or carefully give the command **:e!** *name*CR, which edits the other file discarding the changes you have made to the current file. To have the editor automatically save changes, include *set autowrite* in your EXINIT, and use **:n** instead of **:e**.

5.2. Escaping to a shell

You can get to a shell to execute a single command by giving a *vi* command of the form **:!** *cmd*CR. The system will run the single command *cmd* and when the command finishes, the editor will ask you to hit a RETURN to continue. When you have finished looking at the output on the screen, you should hit RETURN and the editor will clear the screen and redraw it. You can then continue editing. You can also give another **:** command when it asks you for a RETURN; in this case the screen will not be redrawn.

If you wish to execute more than one command in the shell, then you can give the command **:sh**CR. This will give you a new shell, and when you finish with the shell, ending it by typing a **^D**, the editor will clear the screen and continue.

On systems which support it, **^Z** will suspend the editor and return to the (top level) shell. When the editor is resumed, the screen will be redrawn.

5.3. Marking and returning

The command `` returned to the previous place after a motion of the cursor by a command such as **/**, **?** or **G**. You can also mark lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command **m***x*, where you should pick some letter for *x*, say 'a'. Then move the cursor to a different line (any way you like) and hit **`a**. The cursor will return to the place which you marked. Marks last only until you edit another file.

When using operators such as **d** and referring to marked lines, it is often desirable to delete whole lines rather than deleting to the exact position in the line marked by **m**. In this case you can use the form '*x* rather than `*x*. Used without an operator, '*x* will move to the first non-white character of the marked line; similarly '' moves to the first non-white character of the line containing the previous context mark ``.

5.4. Adjusting the screen

If the screen image is messed up because of a transmission error to your terminal, or because some program other than the editor wrote output to your terminal, you can hit a **^L**, the ASCII form-feed character, to cause the screen to be refreshed.

On a dumb terminal, if there are @ lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing **^R** to cause the editor to retype the screen, closing up these holes.

Finally, if you wish to place a certain line on the screen at the top middle or bottom of the screen, you can position the cursor to that line, and then give a **z** command. You should follow the **z** command with a RETURN if you want the line to appear at the top of the window, a **.** if you want it at the center, or a **–** if you want it at the bottom. (**z.**, **z-**, and **z+** are not available on all v2 editors.)

6. Special topics

6.1. Editing on slow terminals

When you are on a slow terminal, it is important to limit the amount of output which is generated to your screen so that you will not suffer long delays, waiting for the screen to be refreshed. We have already pointed out how the editor optimizes the updating of the screen during insertions on dumb terminals to limit the delays, and how the editor erases lines to @ when they are deleted on dumb terminals.

The use of the slow terminal insertion mode is controlled by the *slowopen* option. You can force the editor to use this mode even on faster terminals by giving the command **:se slow**CR. If your system is sluggish this helps lessen the amount of output coming to your terminal. You can disable this option by **:se noslow**CR.

The editor can simulate an intelligent terminal on a dumb one. Try giving the command **:se redraw**CR. This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this by giving the command
:se noredrawCR.

The editor also makes editing more pleasant at low speed by starting editing in a small window, and letting the window expand as you edit. This works particularly well on intelligent terminals. The editor can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try the editor on an intelligent terminal to see how this works.

You can control the size of the window which is redrawn each time the screen is cleared by giving window sizes as argument to the commands which cause large screen motions:

: / ? [[ ]] ` '

Thus if you are searching for a particular instance of a common string in a file you can precede the first search command by a small number, say 3, and the editor will draw three line windows around each instance of the string which it locates.

You can easily expand or contract the window, placing the current line as you choose, by giving a number on a **z** command, after the **z** and before the following RETURN, **.** or – . Thus the command **z5.** redraws the screen with the current line in the center of a five line window.†

If the editor is redrawing or otherwise updating large portions of the display, you can interrupt this updating by hitting a DEL or RUB as usual. If you do this you may partially confuse the editor about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by hitting a **^L**; or move or search again, ignoring the current state of the display.

See section 7.8 on *open* mode for another way to use the *vi* command set on slow terminals.

6.2. Options, set, and editor startup files

The editor has a set of options, some of which have been mentioned above. The most useful options are given in the following table.

Name	Default	Description
autoindent	noai	Supply indentation automatically
autowrite	noaw	Automatic write before **:n**, **:ta**, ^↑, **!**
ignorecase	noic	Ignore case in searching
lisp	nolisp	({) } commands deal with S-expressions
list	nolist	Tabs print as ^I; end of lines marked with $
magic	nomagic	The characters . [and * are special in scans
number	nonu	Lines are displayed prefixed with line numbers
paragraphs	para=IPLPPPQPbpP LI	Macro names which start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one
sections	sect=NHSHH HU	Macro names which start new sections
shiftwidth	sw=8	Shift distance for <, > and input **^D** and **^T**
showmatch	nosm	Show matching (or { as) or } is typed
slowopen	slow	Postpone display updates during inserts
term	dumb	The kind of terminal you are using.

The options are of three kinds: numeric options, string options, and toggle options. You can set numeric and string options by a statement of the form

set *opt*=*val*

and toggle options can be set or unset by statements of one of the forms

set *opt*
set no*opt*

These statements can be placed in your EXINIT in your environment, or given while you are running *vi* by preceding them with a **:** and following them with a CR.

You can get a list of all options which you have changed by the command **:set**CR, or the value of a single option by the command **:set** *opt***?** CR. A list of all possible options and their values is generated by **:set all**CR. Set can be abbreviated **se**. Multiple options can be placed on one line, e.g. **:se ai aw nu**CR.

Options set by the **set** command only last while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands† which are to be run every time you start up *ex*, *edit*, or *vi*. A

† Note that the command **5z.** has an entirely different effect, placing line 5 in the center of a new window.

† All commands which start with **:** are *ex* commands.

typical list includes a **set** command, and possibly a few **map** commands (on v3 editors). Since it is advisable to get these commands on one line, they can be separated with the |character, for example:

> **set** ai aw terse|**map** @ dd|**map** # x

which sets the options *autoindent, autowrite, terse,* (the **set** command), makes @ delete a line, (the first **map**), and makes # delete a character, (the second **map**). (See section 6.9 for a description of the **map** command, which only works in version 3.) This string should be placed in the variable EXINIT in your environment. If you use *csh,* put this line in the file *.login* in your home directory:

> setenv EXINIT ´**set** ai aw terse|**map** @ dd|**map** # x´

If you use the standard v7 shell, put these lines in the file *.profile* in your home directory:

> EXINIT=´**set** ai aw terse|**map** @ dd|**map** # x´
> export EXINIT

On a version 6 system, the concept of environments is not present. In this case, put the line in the file *.exrc* in your home directory.

> **set** ai aw terse|**map** @ dd|**map** # x

Of course, the particulars of the line would depend on which options you wanted to set.

6.3. Recovering lost lines

You might have a serious problem if you delete a number of lines and then regret that they were deleted. Despair not, the editor saves the last 9 deleted blocks of text in a set of numbered registers 1– 9. You can get the *n*'th previous deleted text back in your file by the command "*n***p**. The " here says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and **p** is the put command, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, hit **u** to undo this and then **.** (period) to repeat the put command. In general the **.** command will repeat the last change you made. As a special case, when the last command refers to a numbered text buffer, the **.** command increments the number of the buffer before repeating the command. Thus a sequence of the form

> **"1pu.u.u.**

will, if repeated long enough, show you all the deleted text which has been saved for you. You can omit the **u** commands here to gather up all this text in the buffer, or stop after any **.** command to keep just the then recovered text. The command **P** can also be used rather than **p** to put the recovered text before rather than after the cursor.

6.4. Recovering lost files

If the system crashes, you can recover the work you were doing to within a few changes. You will normally receive mail when you next login giving you the name of the file which has been saved for you. You should then change to the directory where you were when the system crashed and give a command of the form:

> % **vi – r** *name*

replacing *name* with the name of the file which you were editing. This will recover your work to a point near where you left off.†

† In rare cases, some of the lines of the file may be lost. The editor will give you the numbers of these lines and the text of the lines will be replaced by the string 'LOST'. These lines will almost always be among the last few which you changed. You can either choose to discard the changes which you made (if they are easy to remake) or to replace the few lost lines by hand.

You can get a listing of the files which are saved for you by giving the command:

% vi – r

If there is more than one instance of a particular file saved, the editor gives you the newest instance each time you recover it. You can thus get an older saved copy back by first recovering the newer copies.

For this feature to work, *vi* must be correctly installed by a super user on your system, and the *mail* program must exist to receive mail. The invocation "*vi -r*" will not always list all saved files, but they can be recovered even if they are not listed.

6.5. Continuous text input

When you are typing in large amounts of text it is convenient to have lines broken near the right margin automatically. You can cause this to happen by giving the command **:se wm=10**CR. This causes all lines to be broken at a space at least 10 columns from the right hand edge of the screen.*

If the editor breaks an input line and you wish to put it back together you can tell it to join the lines with **J**. You can give **J** a count of the number of lines to be joined as in **3J** to join 3 lines. The editor supplies white space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. You can kill the white space with **x** if you don't want it.

6.6. Features for editing programs

The editor has a number of commands for editing programs. The thing that most distinguishes editing of programs from editing of text is the desirability of maintaining an indented structure to the body of the program. The editor has a *autoindent* facility for helping you generate correctly indented programs.

To enable this facility you can give the command **:se ai**CR. Now try opening a new line with **o** and type some characters on the line after a few tabs. If you now start another line, notice that the editor supplies white space at the beginning of the line to line it up with the previous line. You cannot backspace over this indentation, but you can use **^D** key to backtab over the supplied indentation.

Each time you type **^D** you back up one position, normally to an 8 column boundary. This amount is settable; the editor has an option called *shiftwidth* which you can set to change this value. Try giving the command **:se sw=4**CR and then experimenting with autoindent again.

For shifting lines in the program left and right, there are operators < and >. These shift the lines you specify right or left by one *shiftwidth*. Try << and >> which shift one line left or right, and **<L** and **>L** shifting the rest of the display left and right.

If you have a complicated expression and wish to see how the parentheses match, put the cursor at a left or right parenthesis and hit **%** This will show you the matching parenthesis. This works also for braces { and }, and brackets [and].

If you are editing C programs, you can use the [[and]] keys to advance or retreat to a line starting with a {, i.e. a function declaration at a time. When]] is used with an operator it stops after a line which starts with }; this is sometimes useful with **y**]].

* This feature is not available on some v2 editors. In v2 editors where it is available, the break can only occur to the right of the specified boundary instead of to the left.

6.7. Filtering portions of the buffer

You can run system commands over portions of the buffer using the operator **!**. You can use this to sort lines in the buffer, or to reformat portions of the buffer with a pretty-printer. Try typing in a list of random words, one per line and ending them with a blank line. Back up to the beginning of the list, and then give the command **!}sort**CR. This says to sort the next paragraph of material, and the blank line ends a paragraph.

6.8. Commands for editing LISP†

If you are editing a LISP program you should set the option *lisp* by doing **:se lisp**CR. This changes the **(** and **)** commands to move backward and forward over s-expressions. The **{** and **}** commands are like **(** and **)** but don't stop at atoms. These can be used to skip to the next list, or through a comment quickly.

The *autoindent* option works differently for LISP, supplying indent to align at the first argument to the last open list. If there is no such argument then the indent is two spaces more than the last level.

There is another option which is useful for typing in LISP, the *showmatch* option. Try setting it with **:se sm**CR and then try typing a '(' some words and then a ')'. Notice that the cursor shows the position of the '(' which matches the ')' briefly. This happens only if the matching '(' is on the screen, and the cursor stays there for at most one second.

The editor also has an operator to realign existing lines as though they had been typed in with *lisp* and *autoindent* set. This is the **=** operator. Try the command **=%**at the beginning of a function. This will realign all the lines of the function declaration.

When you are editing LISP,, the [[and]] advance and retreat to lines beginning with a **(**, and are useful for dealing with entire function definitions.

6.9. Macros‡

Vi has a parameterless macro facility, which lets you set it up so that when you hit a single keystroke, the editor will act as though you had hit some longer sequence of keys. You can set this up if you find yourself typing the same sequence of commands repeatedly.

Briefly, there are two flavors of macros:

a) Ones where you put the macro body in a buffer register, say *x*. You can then type @**x** to invoke the macro. The @ may be followed by another @ to repeat the last macro.

b) You can use the *map* command from *vi* (typically in your *EXINIT*) with a command of the form:

   :map *lhs rhs*CR

   mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke (either 1 character or one function key) since it must be entered within one second (unless *notimeout* is set, in which case you can type it as slowly as you wish, and *vi* will wait for you to finish it before it echoes anything). The *lhs* can be no longer than 10 characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs* you should escape them with a **^V**. (It may be necessary to double the **^V** if the map command is given inside *vi*, rather than in *ex.)* Spaces and tabs inside the *rhs* need not be escaped.

   Thus to make the **q** key write and exit the editor, you can give the command

   :map q :wq**^V^V**CR CR

which means that whenever you type **q**, it will be as though you had typed the four characters **:wq**CR. A **^V**'s is needed because without it the CR would end the **:** command, rather than

† The LISP features are not available on some v2 editors due to memory constraints.

‡ The macro feature is available only in version 3 editors.

becoming part of the *map* definition. There are two ^V's because from within *vi*, two ^V's must be typed to get one. The first CR is part of the *rhs*, the second terminates the : command.

Macros can be deleted with

```
unmap lhs
```

If the *lhs* of a macro is "#0" through "#9", this maps the particular function key instead of the 2 character "#" sequence. So that terminals without function keys can access such definitions, the form "#x" will mean function key *x* on all terminals (and need not be typed within one second.) The character "#" can be changed by using a macro in the usual way:

```
:map ^V^V^I #
```

to use tab, for example. (This won't affect the *map* command, which still uses **#**, but just the invocation from visual mode.

The undo command reverses an entire macro call as a unit, if it made any changes.

Placing a '!' after the word **map** causes the mapping to apply to input mode, rather than command mode. Thus, to arrange for ^**T** to be the same as 4 spaces in input mode, you can type:

```
:map ^T ^V₿₿₿₿
```

where ₿ is a blank. The ^V is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

7. Word Abbreviations ‡‡

A feature similar to macros in input mode is word abbreviation. This allows you to type a short word and have it expanded into a longer word or words. The commands are **:abbreviate** and **:unabbreviate** (**:ab** and **:una**) and have the same syntax as **:map**. For example:

```
:ab eecs Electrical Engineering and Computer Sciences
```

causes the word 'eecs' to always be changed into the phrase 'Electrical Engineering and Computer Sciences'. Word abbreviation is different from macros in that only whole words are affected. If 'eecs' were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke, as it should be with a macro.

7.1. Abbreviations

The editor has a number of short commands which abbreviate longer commands which we have introduced here. You can find these commands easily on the quick reference card. They often save a bit of typing and you can learn them as convenient.

8. Nitty-gritty details

8.1. Line representation in the display

The editor folds long logical lines onto many physical lines in the display. Commands which advance lines advance logical lines and will skip over all the segments of a line in one motion. The command | moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Try **80|** on a line which is more than 80 columns long.†

The editor only puts full lines on the display; if there is not enough room on the display to fit a logical line, the editor leaves the physical line empty, placing only an @ on the line as a

‡‡ Version 3 only.
† You can make long lines very easily by using **J** to join together short lines.

place holder. When you delete lines on a dumb terminal, the editor will often just clear the lines to @ to save time (rather than rewriting the rest of the screen.) You can always maximize the information on the screen by giving the **^R** command.

If you wish, you can have the editor place line numbers before each line on the display. Give the command **:se nu**CR to enable this, and the command **:se nonu**CR to turn it off. You can have tabs represented as **^I** and the ends of lines indicated with '$' by giving the command **:se list**CR; **:se nolist**CR turns this off.

Finally, lines consisting of only the character '~' are displayed when the last line in the file is in the middle of the screen. These represent physical lines which are past the logical end of file.

8.2. Counts

Most *vi* commands will use a preceding count to affect their behavior in some way. The following table gives the common ways in which the counts are used:

new window size	**: / ? [[ ]] \` ´**
scroll amount	**^D ^U**
line/column number	**z G |**
repeat effect	most of the rest

The editor maintains a notion of the current default window size. On terminals which run at speeds greater than 1200 baud the editor uses the full terminal screen. On terminals which are slower than 1200 baud (most dialup lines are in this group) the editor uses 8 lines as the default window size. At 1200 baud the default is 16 lines.

This size is the size used when the editor clears and refills the screen after a search or other motion moves far from the edge of the current window. The commands which take a new window size as count all often cause the screen to be redrawn. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen will expand if you move off the top with a – or similar command or off the bottom with a command such as RETURN or **^D**. The window will revert to the last specified size the next time it is cleared and refilled.†

The scroll commands **^D** and **^U** likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus **10a+– – – –** ESC will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands which ignore any counts (such as **^R**), the rest of the editor commands use a count to indicate a simple repetition of their effect. Thus **5w** advances five words on the current line, while **5**RETURN advances five lines. A very useful instance of a count as a repetition is a count given to the **.** command, which repeats the last changing command. If you do **dw** and then **3.**, you will delete first one and then three words. You can then delete two more words with **2.**.

8.3. More file manipulation commands

The following table lists the file manipulation commands which you can use when you are in *vi*. All of these commands are followed by a CR or ESC. The most basic commands are **:w** and **:e**. A normal editing session on a single file will end with a **ZZ** command. If you are editing for a long period of time you can give **:w** commands occasionally after major amounts of editing, and then finish with a **ZZ**. When you edit more than one file, you can finish with one

† But not by a ^L which just redraws the screen as it is.

:w	write back changes
:wq	write and quit
:x	write (if necessary) and quit (same as ZZ).
:e *name*	edit file *name*
:e!	reedit, discarding changes
:e + *name*	edit, starting at end
:e +*n*	edit, starting at line *n*
:e #	edit alternate file
:w *name*	write file *name*
:w! *name*	overwrite file *name*
:*x,y***w** *name*	write lines *x* through *y* to *name*
:r *name*	read file *name* into buffer
:r !*cmd*	read output of *cmd* into buffer
:n	edit next file in argument list
:n!	edit next file, discarding changes to current
:n *args*	specify new argument list
:ta *tag*	edit file containing tag *tag*, at *tag*

with a **:w** and start editing a new file by giving a **:e** command, or set *autowrite* and use **:n** <file>.

If you make changes to the editor's copy of a file, but do not wish to write them back, then you must give an **!** after the command you would otherwise use; this forces the editor to discard any changes you have made. Use this carefully.

The **:e** command can be given a **+** argument to start at the end of the file, or a **+***n* argument to start at line *n*. In actuality, *n* may be any editor command not containing a space, usefully a scan like **+/***pat* or **+?***pat*. In forming new names to the **e** command, you can use the character **%**which is replaced by the current file name, or the character **#** which is replaced by the alternate file name. The alternate file name is generally the last name you typed other than the current file. Thus if you try to do a **:e** and get a diagnostic that you haven't written the file, you can give a **:w** command and then a **:e #** command to redo the previous **:e**.

You can write part of the buffer to a file by finding out the lines that bound the range to be written using ^G, and giving these numbers after the **:** and before the **w**, separated by **,**'s. You can also mark these lines with **m** and then use an address of the form **´***x***,´***y* on the **w** command here.

You can read another file into the buffer after the current line by using the **:r** command. You can similarly read in the output from a command, just use **!***cmd* instead of a file name.

If you wish to edit a set of files in succession, you can give all the names on the command line, and then edit each one in turn using the command **:n**. It is also possible to respecify the list of files to be edited by giving the **:n** command a list of file names, or a pattern to be expanded as you would have given it on the initial *vi* command.

If you are editing large programs, you will find the **:ta** command very useful. It utilizes a data base of function names and their locations, which can be created by programs such as *ctags,* to quickly find a function whose name you give. If the **:ta** command will require the editor to switch files, then you must **:w** or abandon any changes before switching. You can repeat the **:ta** command without any arguments to look for the same tag again. (The tag feature is not available in some v2 editors.)

8.4. More about searching for strings

When you are searching for strings in the file with **/** and **?**, the editor normally places you at the next or previous occurrence of the string. If you are using an operator such as **d**, **c** or **y**, then you may well wish to affect lines up to the line before the line containing the pattern.

You can give a search of the form */pat/– n* to refer to the *n*'th line before the next line containing *pat*, or you can use + instead of – to refer to the lines after the one containing *pat*. If you don't give a line offset, then the editor will affect characters up to the match place, rather than whole lines; thus use "+0" to affect to the line which matches.

You can have the editor ignore the case of words in the searches it does by giving the command **:se ic**CR. The command **:se noic**CR turns this off.

Strings given to searches may actually be regular expressions. If you do not want or need this facility, you should

```
set nomagic
```

in your EXINIT. In this case, only the characters ↑ and $ are special in patterns. The character \ is also then special (as it is most everywhere in the system), and may be used to get at the an extended pattern matching facility. It is also necessary to use a \ before a / in a forward scan or a **?** in a backward scan, in any case. The following table gives the extended forms when **magic** is set.

↑	at beginning of pattern, matches beginning of line
$	at end of pattern, matches end of line
.	matches any character
\<	matches the beginning of a word
\>	matches the end of a word
[*str*]	matches any single character in *str*
[↑*str*]	matches any single character not in *str*
[*x*– *y*]	matches any character between *x* and *y*
*	matches any number of the preceding pattern

If you use **nomagic** mode, then the **.** [and * primitives are given with a preceding \.

8.5. More about input mode

There are a number of characters which you can use to make corrections during input mode. These are summarized in the following table.

^H	deletes the last input character
^W	deletes the last input word, defined as by **b**
erase	your erase character, same as **^H**
kill	your kill character, deletes the input on this line
\	escapes a following **^H** and your erase and kill
ESC	ends an insertion
DEL	interrupts an insertion, terminating it abnormally
CR	starts a new line
^D	backtabs over *autoindent*
0^D	kills all the *autoindent*
↑^D	same as **0^D**, but restores indent next line
^V	quotes the next non-printing character into the file

The most usual way of making corrections to input is by typing **^H** to correct a single character, or by typing one or more **^W**'s to back over incorrect words. If you use # as your erase character in the normal system, it will work like **^H**.

Your system kill character, normally @ , **^X** or **^U**, will erase all the input you have given on the current line. In general, you can neither erase input back around a line boundary nor can you erase characters which you did not insert with this insertion command. To make corrections on the previous line after a new line has been started you can hit ESC to end the insertion, move over and make the correction, and then return to where you were to continue.

The command **A** which appends at the end of the current line is often useful for continuing.

If you wish to type in your erase or kill character (say # or @) then you must precede it with a \\, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a **^V**. The **^V** echoes as a ↑ character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact you may type any character and it will be inserted into the file at that point.*

If you are using *autoindent* you can backtab over the indent which it supplies by typing a **^D**. This backs up to a *shiftwidth* boundary. This only works immediately after the supplied *autoindent.*

When you are using *autoindent* you may wish to place a label at the left margin of a line. The way to do this easily is to type ↑ and then **^D**. The editor will move the cursor to the left margin for one line, and restore the previous indent on the next. You can also type a **0** followed immediately by a **^D** if you wish to kill all the indent and not have it come back on the next line.

8.6. Upper case only terminals

If your terminal has only upper case, you can still use *vi* by using the normal system convention for typing on such a terminal. Characters which you normally type are converted to lower case, and you can type upper case letters by preceding them with a \\. The characters { ~ } | ` are not available on such terminals, but you can escape them as \\(\\↑ \\) \\! \\'. These characters are represented on the display in the same way they are typed.‡ ‡

8.7. Vi and ex

Vi is actually one mode of editing within the editor *ex*. When you are running *vi* you can escape to the line oriented editor of *ex* by giving the command **Q**. All of the **:** commands which were introduced above are available in *ex*. Likewise, most *ex* commands can be invoked from *vi* using :. Just give them without the **:** and follow them with a CR.

In rare instances, an internal error may occur in *vi*. In this case you will get a diagnostic and be left in the command mode of *ex*. You can then save your work and quit if you wish by giving a command **x** after the **:** which *ex* prompts you with, or you can reenter *vi* by giving *ex* a *vi* command.

There are a number of things which you can do more easily in *ex* than in *vi*. Systematic changes in line oriented material are particularly easy. You can read the advanced editing documents for the editor *ed* to find out a lot more about this style of editing. Experienced users often mix their use of *ex* command mode and *vi* command mode to speed the work they are doing.

8.8. Open mode: vi on hardcopy terminals and "glass tty's" ‡

If you are on a hardcopy terminal or a terminal which does not have a cursor which can move off the bottom line, you can still use the command set of *vi*, but in a different mode. When you give a *vi* command, the editor will tell you that it is using *open* mode. This name comes from the *open* command in *ex*, which is used to get into the same mode.

The only difference between *visual* mode and *open* mode is the way in which the text is

* This is not quite true. The implementation of the editor does not allow the NULL (^@) character to appear in files. Also the LF (linefeed or ^J) character is used by the editor to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ↑ before you type the character. In fact, the editor will treat a following letter as a request for the corresponding control character. This is the only way to type ^S or ^Q, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

‡ The \\ character you give will not echo until you type another key.

‡ Not available in all v2 editors due to memory constraints.

displayed.

In *open* mode the editor uses a single line window into the file, and moving backward and forward in the file causes new lines to be displayed, always below the current line. Two commands of *vi* work differently in *open:* **z** and **^R**. The **z** command does not take parameters, but rather draws a window of context around the current line and then returns you to the current line.

If you are on a hardcopy terminal, the **^R** command will retype the current line. On such terminals, the editor normally uses two lines to represent the current line. The first line is a copy of the line as you started to edit it, and you work on the line below this line. When you delete characters, the editor types a number of \'s to show you the characters which are deleted. The editor also reprints the current line soon after such changes so that you can see what the line looks like again.

It is sometimes useful to use this mode on very slow terminals which can support *vi* in the full screen mode. You can do this by entering *ex* and using an *open* command.

Acknowledgements

Bruce Englar encouraged the early development of this display editor. Peter Kessler helped bring sanity to version 2's command layout. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and Unix systems.

Vi Appendix

Appendix: character functions

This appendix gives the uses the editor makes of each character. The characters are presented in their order in the ASCII character set: Control characters come first, then most special characters, then the digits, upper and then lower case characters.

For each character we tell a meaning it has as a command and any meaning it has during an insert. If it has only meaning as a command, then only this is discussed. Section numbers in parentheses indicate where the character is discussed; a 'f' after the section number means that the character is mentioned in a footnote.

^@	Not a command character. If typed as the first character of an insertion it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted the mechanism is not available. A ^@ cannot be part of the file due to the editor implementation (7.5f).
^A	Unused.
^B	Backward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).
^C	Unused.
^D	As a command, scrolls down a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future **^D** and **^U** commands (2.1, 7.2). During an insert, backtabs over *autoindent* white space at the beginning of a line (6.6, 7.5); this white space cannot be backspaced over.
^E	Exposes one more line below the current screen in the file, leaving the cursor where it is if possible. (Version 3 only.)
^F	Forward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).
^G	Equivalent to **:f**CR, printing the current file, whether it has been modified, the current line number and the number of lines in the file, and the percentage of the way through the file that you are.
^H (BS)	Same as **left arrow**. (See **h**). During an insert, eliminates the last input character, backing over it but not erasing it; it remains so you can see what you typed if you wish to type something only slightly different (3.1, 7.5).
^I (TAB)	Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character it rests at the last of the spaces which represent the tab. The spacing of tabstops is controlled by the *tabstop* option (4.1, 6.6).
^J (LF)	Same as **down arrow** (see **j**).
^K	Unused.
^L	The ASCII formfeed character, this causes the screen to be cleared and redrawn. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt (5.4, 7.2f).
^M (CR)	A carriage return advances to the next line, at the first non-white position in the line. Given a count, it advances that many lines (2.3). During an insert, a CR causes the insert to continue onto another line (3.1).
^N	Same as **down arrow** (see **j**).

^O	Unused.
^P	Same as **up arrow** (see **k**).
^Q	Not a command character. In input mode, **^Q** quotes the next character, the same as **^V**, except that some teletype drivers will eat the **^Q** so that the editor never sees it.
^R	Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line (5.4, 7.2, 7.8).
^S	Unused. Some teletype drivers use **^S** to suspend output until **^Q**is
^T	Not a command character. During an insert, with *autoindent* set and at the beginning of the line, inserts *shiftwidth* white space.
^U	Scrolls the screen up, inverting **^D** which scrolls down. Counts work as they do for **^D**, and the previous scroll amount is common to both. On a dumb terminal, **^U** will often necessitate clearing and redrawing the screen further back in the file (2.1, 7.2).
^V	Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file (4.2, 7.5).
^W	Not a command character. During an insert, backs up as **b** would in command mode; the deleted characters remain on the display (see **^H**) (7.5).
^X	Unused.
^Y	Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; however, it is next to **^U** which scrolls up a bunch.) (Version 3 only.)
^Z	If supported by the Unix system, stops the editor, exiting to the top level shell. Same as **:stop**CR. Otherwise, unused.
^[(ESC)	Cancels a partially formed command, such as a **z** when no following character has yet been given; terminates inputs on the last line (read by commands such as **:** / and **?**); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. You can thus hit ESC if you don't know what is happening till the editor rings the bell. If you don't know if you are in insert mode you can type ESC**a**, and then material to be input; the material will be inserted correctly whether or not you were in insert mode when you started (1.5, 3.1, 7.5).
^\	Unused.
^]	Searches for the word which is after the cursor as a tag. Equivalent to typing **:ta**, this word, and then a CR. Mnemonically, this command is "go right to" (7.3).
^↑	Equivalent to **:e #**CR, returning to the previous position in the last edited file, or editing a file which you specified if you got a 'No write since last change diagnostic' and do not want to have to type the file name again (7.3). (You have to do a **:w** before ^↑ will work in this case. If you do not wish to write the file you should do **:e! #**CR instead.)
^_	Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.
SPACE	Same as **right arrow** (see **l**).
!	An operator, which processes lines from the buffer with reformatting commands. Follow **!** with the object to be processed, and then the command name terminated by CR. Doubling **!** and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the **!**. Thus

	2!}*fmt*CR reformats the next two paragraphs by running them through the program *fmt.* If you are working on LISP, the command **!**%*grind*CR,* given at the beginning of a function, will run the text of the function through the LISP grinder (6.7, 7.3). To read a file or the output of a command into the buffer use **:r** (7.3). To simply execute a command use **:!** (7.3).
"	Precedes a named buffer specification. There are named buffers **1**– **9** used for saving deleted text and named buffers **a**– **z** into which you can place text (4.3, 6.3)
#	The macro character which, when followed by a number, will substitute for a function key on terminals without function keys (6.9). In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a \ to insert it, since it normally backs over the last input character you gave.
$	Moves to the end of the current line. If you **:se list**CR, then the end of each line will be shown by printing a **$** after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus **2$** advances to the end of the following line.
%	Moves to the parenthesis or brace { } which balances the parenthesis or brace at the current cursor position.
&	A synonym for **:&**CR, by analogy with the *ex* **&** command.
´	When followed by a ´ returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a**– **z**, returns to the line which was marked with this letter with a **m** command, at the first non-white character in the line. (2.2, 5.3). When used with an operator such as **d**, the operation takes place over complete lines; if you use `, the operation takes place from the exact marked place to the current cursor position within the line.
(	Retreats to the beginning of a sentence, or to the beginning of a LISP s-expression if the *lisp* option is set. A sentence ends at a **. !** or **?** which is followed by either the end of a line or by two spaces. Any number of closing **)**] " and ´ characters may appear after the **. !** or **?**, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see { and [[below). A count advances that many sentences (4.2, 6.8).
)	Advances to the beginning of a sentence. A count repeats the effect. See (above for the definition of a sentence (4.2, 6.8).
*	Unused.
+	Same as CR when used as a command.
,	Reverse of the last **f F t** or **T** command, looking the other way in the current line. Especially useful after hitting too many **;** characters. A count repeats the search.
–	Retreats to the previous line at the first non-white character. This is the inverse of **+** and RETURN. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if this is not possible. If a large amount of scrolling would be required the screen is also cleared and redrawn, with the current line at the center (2.3).

*Both *fmt* and *grind* are Berkeley programs and may not be present at all installations.

.	Repeats the last command which changed the buffer. Especially useful when deleting words or lines; you can delete some words/lines and then hit **.** to delete more and more words/lines. Given a count, it passes it on to the command being repeated. Thus after a **2dw**, **3.** deletes three words (3.3, 6.3, 7.2, 7.4).
/	Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input on the bottom line; an returns to command state without ever searching. The search begins when you hit CR to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress; the search may then be terminated with a DEL or RUB, or by backspacing when at the beginning of the bottom line, returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer. When used with an operator the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern you can force whole lines to be affected. To do this give a pattern with a closing a closing / and then an offset $+n$ or $-n$. To include the character / in the search string, you must escape it with a preceding \\. A ↑ at the beginning of the pattern forces the match to occur at the beginning of a line only; this speeds the search. A $ at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available, see section 7.4; unless you set **nomagic** in your *.exrc* file you will have to preceed the characters **.** [* and ˜ in the search pattern with a \\ to get them to work as you would naively expect (1.5, 2,2, 6.1, 7.2, 7.4).
0	Moves to the first character on the current line. Also used, in forming numbers, after an initial **1**– **9**.
1– **9**	Used to form numeric arguments to commands (2.3, 7.2).
:	A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with an CR, and the command then executed. You can return to where you were by hitting DEL or RUB if you hit **:** accidentally (see primarily 6.2 and 7.3).
;	Repeats the last single character find which used **f F t** or **T**. A count iterates the basic scan (4.1).
<	An operator which shifts lines left one *shiftwidth*, normally 8 spaces. Like all operators, affects lines when repeated, as in <<. Counts are passed through to the basic object, thus **3**<< shifts three lines (6.6, 7.2).
=	Reindents line for LISP, as though they were typed in with *lisp* and *autoindent* set (6.8).
>	An operator which shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in >>. Counts repeat the basic object (6.6, 7.2).
?	Scans backwards, the opposite of /. See the / description above for details on scanning (2.2, 6.1, 7.4).
@	A macro character (6.9). If this is your kill character, you must escape it with a \\ to type it in during input mode, as it normally backs over the input you have given on the current line (3.1, 3.4, 7.5).
A	Appends at the end of line, a synonym for **$a** (7.2).

B	Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect (2.4).
C	Changes the rest of the text on the current line; a synonym for **c$**.
D	Deletes the rest of the text on the current line; a synonym for **d$**.
E	Moves forward to the end of a word, defined as blanks and non-blanks, like **B** and **W**. A count repeats the effect.
F	Finds a single following character, backwards in the current line. A count repeats this search that many times (4.1).
G	Goes to the line number given as preceding argument, or the end of the file if no preceding count is given. The screen is redrawn with the new current line in the center if necessary (7.2).
H	**Home arrow**. Homes the cursor to the top line on the screen. If a count is given, then the cursor is moved to the count'th line on the screen. In any case the cursor is moved to the first non-white character on the line. If used as the target of an operator, full lines are affected (2.3, 3.2).
I	Inserts at the beginning of a line; a synonym for **↑i**.
J	Joins together lines, supplying appropriate white space: one space between words, two spaces after a **.**, and no spaces at all if the first character of the joined on line is **)**. A count causes that many lines to be joined rather than the default two (6.5, 7.1f).
K	Unused.
L	Moves the cursor to the first non-white character of the last line on the screen. With a count, to the first non-white of the count'th line from the bottom. Operators affect whole lines when used with **L** (2.3).
M	Moves the cursor to the middle line on the screen, at the first non-white position on the line (2.3).
N	Scans for the next match of the last pattern given to **/** or **?**, but in the reverse direction; this is the reverse of **n**.
O	Opens a new line above the current line and inputs text there up to an ESC. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the *slowopen* option works better (3.1).
P	Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted between the characters before and at the cursor. May be preceded by a named buffer specification "*x* to retrieve the contents of the buffer; buffers **1**– **9** contain deleted material, buffers **a**– **z** are available for general use (6.3).
Q	Quits from *vi* to *ex* command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the **:** commands; the editor supplies the **:** as a prompt (7.7).
R	Replaces characters on the screen with characters you type (overlay fashion). Terminates with an ESC.
S	Changes whole lines, a synonym for **cc**. A count substitutes for that many lines. The lines are saved in the numeric buffers, and erased on the screen before the substitution begins.
T	Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as **d** (4.1).

U	Restores the current line to its state before you started changing it (3.5).
V	Unused.
W	Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count repeats the effect (2.4).
X	Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.
Y	Yanks a copy of the current line into the unnamed buffer, to be put back by a later **p** or **P**; a very useful synonym for **yy**. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer (7.4).
ZZ	Exits the editor. (Same as **:x**CR.) If any changes have been made, the buffer is written out to the current file. Then the editor quits.
[[	Backs up to the previous section boundary. A section begins at each macro in the *sections* option, normally a '.NH' or '.SH' and also at lines which which start with a formfeed **^L**. Lines beginning with { also stop [[; this makes it useful for looking backwards, a function at a time, in C programs. If the option *lisp* is set, stops at each (at the beginning of a line, and is thus useful for moving backwards at the top level LISP objects. (4.2, 6.1, 6.6, 7.2).
\	Unused.
]]	Forward to a section boundary, see [[for a definition (4.2, 6.1, 6.6, 7.2).
↑	Moves to the first non-white position on the current line (4.4).
_	Unused.
`	When followed by a ` returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a**– **z**, returns to the position which was marked with this letter with a **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; if you use ', the operation takes place over complete lines (2.2, 5.3).
a	Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with an ESC (3.1, 7.2).
b	Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect (2.4).
c	An operator which changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text which is changed away is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed away is marked with a $. A count causes that many objects to be affected, thus both **3c)** and **c3)** change the following three sentences (7.4).
d	An operator which deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w** (3.3, 3.4, 4.1, 7.4).
e	Advances to the end of the next word, defined as for **b** and **w**. A count repeats the effect (2.4, 3.1).
f	Finds the first instance of the next character following the cursor on the current line. A count repeats the find (4.1).

g	Unused.
	Arrow keys **h**, **j**, **k**, **l**, and **H**.
h	**Left arrow**. Moves the cursor one character to the left. Like the other arrow keys, either **h**, the **left arrow** key, or one of the synonyms (**^H**) has the same effect. On v2 editors, arrow keys on certain kinds of terminals (those which send escape sequences, such as vt52, c100, or hp) cannot be used. A count repeats the effect (3.1, 7.5).
i	Inserts text before the cursor, otherwise like **a** (7.2).
j	**Down arrow**. Moves the cursor one line down in the same column. If the position does not exist, *vi* comes as close as possible to the same column. Synonyms include **^J** (linefeed) and **^N**.
k	**Up arrow**. Moves the cursor one line up. **^P** is a synonym.
l	**Right arrow**. Moves the cursor one character to the right. SPACE is a synonym.
m	Marks the current position of the cursor in the mark register which is specified by the next character **a**– **z**. Return to this position or use with an operator using ` or ´ (5.3).
n	Repeats the last **/** or **?** scanning commands (2.2).
o	Opens new lines below the current line; otherwise like **O** (3.1).
p	Puts text after/below the cursor; otherwise like **P** (6.3).
q	Unused.
r	Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R** above which is the more usually useful iteration of **r** (3.2).
s	Changes the single character under the cursor to the text which follows up to an ESC; given a count, that many characters from the current line are changed. The last character to be changed is marked with **$** as in **c** (3.2).
t	Advances the cursor upto the character before the next character typed. Most useful with operators such as **d** and **c** to delete the characters up to a following character. You can use **.** to delete more if this doesn't delete enough the first time (4.1).
u	Undoes the last change made to the current buffer. If repeated, will alternate between these two states, thus is its own inverse. When used after an insert which inserted text on more than one line, the lines are saved in the numeric named buffers (3.5).
v	Unused.
w	Advances to the beginning of the next word, as defined by **b** (2.4).
x	Deletes the single character under the cursor. With a count deletes deletes that many characters forward from the cursor position, but only on the current line (6.5).
y	An operator, yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, "*x*, the text is placed in that buffer also. Text can be recovered by a later **p** or **P** (7.4).
z	Redraws the screen with the current line placed as specified by the following character: RETURN specifies the top of the screen, **.** the center of the screen, and **–** at the bottom of the screen. A count may be given after the **z** and before the following character to specify the new screen size for the redraw. A

	count before the **z** gives the number of the line to place in the center of the screen instead of the default current line. (5.4)
{	Retreats to the beginning of the beginning of the preceding paragraph. A paragraph begins at each macro in the ***paragraphs*** option, normally '.IP', '.LP', '.PP', '.QP' and '.bp'. A paragraph also begins after a completely empty line, and at each section boundary (see [[above) (4.2, 6.8, 7.6).
|	Places the cursor on the character in the column specified by the count (7.1, 7.2).
}	Advances to the beginning of the next paragraph. See { for the definition of paragraph (4.2, 6.8, 7.6).
~	Unused.
^? (DEL)	Interrupts the editor, returning it to command accepting state (1.5, 7.5)

Vi Command & Function Reference

This document is based on a paper by Alan Hewett and Mark Horton of the University of California, Berkeley. Revised for version 2.12.

1. Disclaimer

This document is a *partial* listing of **vi** capabilities; probably the most common or useful. Experimentation is recommended.

2. Notation

[**option**] is used to denote optional parts of a command. Many **vi** commands have an optional count. [**cnt**] means that an optional number may precede the command to multiply or iterate the command. {**variable item**} is used to denote parts of the command which must appear, but can take a number of different values. <**character** [**-character**]> means that the character or one of the characters in the range described between the two angle brackets is to be typed. For example <**esc**> means the **escape** key is to be typed. <**a-z**> means that a lower case letter is to be typed. ^<**character**> means that the character is to be typed as a **control** character, that is, with the <**cntl**> key held down while simultaneously typing the specified character. In this document control characters will be denoted using the *upper case* character, but ^<uppercase chr> and ^<lowercase chr> are equivalent. That is, for example, <^**D**> is equal to <^**d**>. The most common character abbreviations used in this list are as follows:

<esc>	escape, octal 033
<cr>	carriage return, ^M, octal 015
<lf>	linefeed ^J, octal 012
<nl>	newline, ^J, octal 012 (same as linefeed)
<bs>	backspace, ^H, octal 010
<tab>	tab, ^I, octal 011
<bell>	bell, ^G, octal 07
<ff>	formfeed, ^L, octal 014
<sp>	space, octal 040
<del>	delete, octal 0177

3. Basics

To run **vi** the shell variable **TERM** must be defined and exported to your environment. How you do this depends on which shell you are using. You can tell which shell you have by the character it prompts you for commands with. The Bourne shell prompts with '$', and the C shell prompts with '%'. For these examples, we will suppose that you are using an HP 2621 terminal, whose termcap name is "2621".

3.1. Bourne Shell

To manually set your terminal type to 2621 you would type:

```
TERM=2621
export TERM
```

There are various ways of having this automatically or semi-automatically done when you log in. Suppose you usually dial in on a 2621. You want to tell this to the machine, but still have it work when you use a hardwired terminal. The recommended way, if you have the **tset** program, is to use the sequence

```
tset - s - d 2621 > tset$$
. tset$$
rm tset$$
```

in your .login (for csh) or the same thing using '.' instead of 'source' in your .profile (for sh). The above line says that if you are dialing in you are on a 2621, but if you are on a hardwired terminal it figures out your terminal type from an on-line list.

3.2. The C Shell

To manually set your terminal type to 2621 you would type:

```
setenv TERM 2621
```

There are various ways of having this automatically or semi-automatically done when you log in. Suppose you usually dial in on a 2621. You want to tell this to the machine, but still have it work when you use a hardwired terminal. The recommended way, if you have the **tset** program, is to use the sequence

```
tset - s - d 2621 > tset$$
source tset$$
rm tset$$
```

in your .login.* The above line says that if you are dialing in you are on a 2621, but if you are on a hardwired terminal it figures out your terminal type from an on-line list.

4. Normal Commands

Vi is a visual editor with a window on the file. What you see on the screen is **vi**'s current notion of what your file will contain, (at this point in the file), when it is written out. Most commands do not cause any change in the screen until the complete command is typed. Should you get confused while typing a command, you can abort the command by typing an <del> character. You will know you are back to command level when you hear a <bell>. Usually typing an <esc> will produce the same result. When **vi** gets an improperly formatted command it rings the <bell>. Following are the **vi** commands broken down by function.

4.1. Entry and Exit

To enter **vi** on a particular *file*, type

vi *file*

The file will be read in and the cursor will be placed at the beginning of the first line. The first screenfull of the file will be displayed on the terminal.

To get out of the editor, type

```
ZZ
```

If you are in some special mode, such as input mode or the middle of a multi-keystroke command, it may be necessary to type <esc> first.

4.2. Cursor and Page Motion

NOTE: The arrow keys (see the next four commands) on certain kinds of terminals will not work with the PDP-11 version of vi. The control versions or the hjkl versions will work on any terminal. Experienced users prefer the hjkl keys because they are always right under their fingers. Beginners often prefer the arrow keys, since they do not require memorization of which hjkl key is which. The mnemonic value of hjkl is clear from looking at the keyboard of

* On a version 6 system without environments, the invocation of tset is simpler, just add the line "tset - d 2621" to your .login or .profile.

an adm3a.

[cnt]<bs> or [cnt]h or [cnt]←
: Move the cursor to the left one character. Cursor stops at the left margin of the page. If cnt is given, these commands move that many spaces.

[cnt]^N or [cnt]j or [cnt]↓ or [cnt]<lf>
: Move down one line. Moving off the screen scrolls the window to force a new line onto the screen. Mnemonic: **N**ext

[cnt]^P or [cnt]k or [cnt]↑
: Move up one line. Moving off the top of the screen forces new text onto the screen. Mnemonic: **P**revious

[cnt]<sp> or [cnt]l or [cnt]→
: Move to the right one character. Cursor will not go beyond the end of the line.

[cnt]- Move the cursor up the screen to the beginning of the next line. Scroll if necessary.

[cnt]+ or [cnt]<cr>

: Move the cursor down the screen to the beginning of the next line. Scroll up if necessary.

[cnt]$ Move the cursor to the end of the line. If there is a count, move to the end of the line "cnt" lines forward in the file.

^ Move the cursor to the beginning of the first word on the line.

0 Move the cursor to the left margin of the current line.

[cnt]| Move the cursor to the column specified by the count. The default is column zero.

[cnt]w Move the cursor to the beginning of the next word. If there is a count, then move forward that many words and position the cursor at the beginning of the word. Mnemonic: next-**w**ord

[cnt]W Move the cursor to the beginning of the next word which follows a "white space" (<sp>,<tab>, or <nl>). Ignore other punctuation.

[cnt]b Move the cursor to the preceding word. Mnemonic: **b**ackup-word

[cnt]B Move the cursor to the preceding word that is separated from the current word by a "white space" (<sp>,<tab>, or <nl>).

[cnt]e Move the cursor to the end of the current word or the end of the "cnt"'th word hence. Mnemonic: **e**nd-of-word

[cnt]E Move the cursor to the end of the current word which is delimited by "white space" (<sp>,<tab>, or <nl>).

[line number]G
: Move the cursor to the line specified. Of particular use are the sequences "1G" and "G", which move the cursor to the beginning and the end of the file respectively. Mnemonic: **G**o-to

NOTE: The next four commands (^D, ^U, ^F, ^B) are not true motion commands, in that they cannot be used as the object of commands such as delete or change.

[cnt]^D Move the cursor down in the file by "cnt" lines (or the last "cnt" if a new count isn't given. The initial default is half a page.) The screen is simultaneously scrolled up. Mnemonic: **D**own

[cnt] ^U	Move the cursor up in the file by "cnt" lines. The screen is simultaneously scrolled down. Mnemonic: **U**p
[cnt] ^F	Move the cursor to the next page. A count moves that many pages. Two lines of the previous page are kept on the screen for continuity if possible. Mnemonic: **F**orward-a-page
[cnt] ^B	Move the cursor to the previous page. Two lines of the current page are kept if possible. Mnemonic: **B**ackup-a-page
[cnt] (	Move the cursor to the beginning of the next sentence. A sentence is defined as ending with a ".", "!", or "?" followed by two spaces or a <nl>.
[cnt])	Move the cursor backwards to the beginning of a sentence.
[cnt] }	Move the cursor to the beginning of the next paragraph. This command works best inside **nroff** documents. It understands two sets of **nroff** macros, – **ms** and – **mm**, for which the commands ".IP", ".LP", ".PP", ".QP", "P", as well as the nroff command ".bp" are considered to be paragraph delimiters. A blank line also delimits a paragraph. The **nroff** macros that it accepts as paragraph delimiters is adjustable. See **paragraphs** under the **Set Commands** section.
[cnt] {	Move the cursor backwards to the beginning of a paragraph.
]]	Move the cursor to the next "section", where a section is defined by two sets of **nroff** macros, – **ms** and – **mm**, in which ".NH", ".SH", and ".H" delimit a section. A line beginning with a <ff><nl> sequence, or a line beginning with a "{" are also considered to be section delimiters. The last option makes it useful for finding the beginnings of C functions. The **nroff** macros that are used for section delimiters can be adjusted. See **sections** under the **Set Commands** section.
[[	Move the cursor backwards to the beginning of a section.
%	Move the cursor to the matching parenthesis or brace. This is very useful in C or lisp code. If the cursor is sitting on a ( ) { or } the cursor is moved to the matching character at the other end of the section. If the cursor is not sitting on a brace or a parenthesis, **vi** searches forward until it finds one and then jumps to the match mate.
[cnt] H	If there is no count move the cursor to the top left position on the screen. If there is a count, then move the cursor to the beginning of the line "cnt" lines from the top of the screen. Mnemonic: **H**ome
[cnt] L	If there is no count move the cursor to the beginning of the last line on the screen. If there is a count, then move the cursor to the beginning of the line "cnt" lines from the bottom of the screen. Mnemonic: **L**ast
M	Move the cursor to the beginning of the middle line on the screen. Mnemonic: **M**iddle
m<a-z>	This command does not move the cursor, but it **marks** the place in the file and the character "<a-z>" becomes the label for referring to this location in the file. See the next two commands. Mnemonic: **m**ark **NOTE:** The mark command is not a motion, and cannot be used as the target of commands such as delete.
´<a-z>	Move the cursor to the beginning of the line that is marked with the label "<a-z>".
`<a-z>	Move the cursor to the exact position on the line that was marked with with the label "<a-z>".

''	Move the cursor back to the beginning of the line where it was before the last "non-relative" move. A "non-relative" move is something such as a search or a jump to a specific line in the file, rather than moving the cursor or scrolling the screen.
``	Move the cursor back to the exact spot on the line where it was located before the last "non-relative" move.

4.3. Searches

The following commands allow you to search for items in a file.

[cnt]f{chr}	Search forward on the line for the next or "cnt"'th occurrence of the character "chr". The cursor is placed **at** the character of interest. Mnemonic: **f**ind character
[cnt]F{chr}	Search backwards on the line for the next or "cnt"'th occurrence of the character "chr". The cursor is placed **at** the character of interest.
[cnt]t{chr}	Search forward on the line for the next or "cnt"'th occurrence of the character "chr". The cursor is placed **just preceding** the character of interest. Mnemonic: move cursor up **t**o character
[cnt]T{chr}	Search backwards on the line for the next or "cnt"'th occurrence of the character "chr". The cursor is placed **just preceding** the character of interest.
[cnt];	Repeat the last "f", "F", "t" or "T" command.
[cnt],	Repeat the last "f", "F", "t" or "T" command, but in the opposite search direction. This is useful if you overshoot.
[cnt]/[string]/<nl>	Search forward for the next occurrence of "string". Wrap around at the end of the file does occur. The final </> is not required.
[cnt]?[string]?<nl>	Search backwards for the next occurrence of "string". If a count is specified, the count becomes the new window size. Wrap around at the beginning of the file does occur. The final <**?**> is not required.
n	Repeat the last /[string]/ or ?[string]? search. Mnemonic: **n**ext occurrence.
N	Repeat the last /[string]/ or ?[string]? search, but in the reverse direction.
:g/[string]/[editor command]<nl>	Using the **:** syntax it is possible to do global searches ala the standard UNIX "ed" editor.

4.4. Text Insertion

The following commands allow for the insertion of text. All multicharacter text insertions are terminated with an <esc> character. The last change can always be **undone** by typing a **u**. The text insert in insertion mode can contain newlines.

a{text}<esc>	Insert text immediately following the cursor position. Mnemonic: **a**ppend
A{text}<esc>	Insert text at the end of the current line. Mnemonic: **A**ppend
i{text}<esc>	Insert text immediately preceding the cursor position. Mnemonic: **i**nsert
I{text}<esc>	Insert text at the beginning of the current line.
o{text}<esc>	Insert a new line after the line on which the cursor appears and insert text there. Mnemonic: **o**pen new line
O{text}<esc>	Insert a new line preceding the line on which the cursor appears and insert text there.

4.5. Text Deletion

The following commands allow the user to delete text in various ways. All changes can always be **undone** by typing the **u** command.

[cnt]x	Delete the character or characters starting at the cursor position.
[cnt]X	Delete the character or characters starting at the character preceding the cursor position.
D	Deletes the remainder of the line starting at the cursor. Mnemonic: **D**elete the rest of line
[cnt] d{motion}	Deletes one or more occurrences of the specified motion. Any motion from sections 4.1 and 4.2 can be used here. The d can be stuttered (e.g. [cnt] dd) to delete cnt lines.

4.6. Text Replacement

The following commands allow the user to simultaneously delete and insert new text. All such actions can be **undone** by typing **u** following the command.

r<chr>	Replaces the character at the current cursor position with <chr>. This is a one character replacement. No <esc> is required for termination. Mnemonic: **r**eplace character
R{text}<esc>	Starts overlaying the characters on the screen with whatever you type. It does not stop until an <esc> is typed.
[cnt] s{text}<esc>	Substitute for "cnt" characters beginning at the current cursor position. A "$" will appear at the position in the text where the "cnt"'th character appears so you will know how much you are erasing. Mnemonic: **s**ubstitute
[cnt] S{text}<esc>	Substitute for the entire current line (or lines). If no count is given, a "$" appears at the end of the current line. If a count of more than 1 is given, all the lines to be replaced are deleted before the insertion begins.
[cnt] c{motion}{text}<esc>	Change the specified "motion" by replacing it with the insertion text. A "$" will appear at the end of the last item that is being deleted unless the deletion involves whole lines. Motion's can be any motion from sections 4.1 or 4.2. Stuttering the c (e.g. [cnt] cc) changes cnt lines.

4.7. Moving Text

Vi provides a number of ways of moving chunks of text around. There are nine buffers into which each piece of text which is deleted or "yanked" is put in addition to the "undo" buffer. The most recent deletion or yank is in the "undo" buffer and also usually in buffer 1, the next most recent in buffer 2, and so forth. Each new deletion pushes down all the older deletions. Deletions older than 9 disappear. There is also a set of named registers, a-z, into which text can optionally be placed. If any delete or replacement type command is preceded by "<**a-z**>, that named buffer will contain the text deleted after the command is executed. For example,

"a3dd will delete three lines starting at the current line and put them in buffer **"a.*** There are two more basic commands and some variations useful in getting and putting text into a file.

["<a-z>][cnt]y{motion}

Yank the specified item or "cnt" items and put in the "undo" buffer or the specified buffer. The variety of "items" that can be yanked is the same as those that can be deleted with the "d" command or changed with the "c" command. In the same way that "dd" means delete the current line and "cc" means replace the current line, "yy" means yank the current line.

["<a-z>][cnt]Y Yank the current line or the "cnt" lines starting from the current line. If no buffer is specified, they will go into the "undo" buffer, like any delete would. It is equivalent to "yy". Mnemonic: **Y**ank

["<a-z>]p Put "undo" buffer or the specified buffer down **after** the cursor. If whole lines were yanked or deleted into the buffer, then they will be put down on the line following the line the cursor is on. If something else was deleted, like a word or sentence, then it will be inserted immediately following the cursor. Mnemonic: **p**ut buffer

It should be noted that text in the named buffers remains there when you start editing a new file with the **:e file<esc>** command. Since this is so, it is possible to copy or delete text from one file and carry it over to another file in the buffers. However, the undo buffer and the ability to undo are lost when changing files.

["<a-z>]P Put "undo" buffer or the specified buffer down **before** the cursor. If whole lines where yanked or deleted into the buffer, then they will be put down on the line preceding the line the cursor is on. If something else was deleted, like a word or sentence, then it will be inserted immediately preceding the cursor.

[cnt]>{motion} The shift operator will right shift all the text from the line on which the cursor is located to the line where the **motion** is located. The text is shifted by one **shiftwidth**. (See section 6.) >> means right shift the current line or lines.

[cnt]<{motion} The shift operator will left shift all the text from the line on which the cursor is located to the line where the **item** is located. The text is shifted by one **shiftwidth**. (See section 6.) << means left shift the current line or lines. Once the line has reached the left margin it is not further affected.

[cnt]={motion} Prettyprints the indicated area according to **lisp** conventions. The area should be a lisp s-expression.

4.8. Miscellaneous Commands

Vi has a number of miscellaneous commands that are very useful. They are:

ZZ This is the normal way to exit from vi. If any changes have been made, the file is written out. Then you are returned to the shell.

^L Redraw the current screen. This is useful if someone "write"s you while you are in "vi" or if for any reason garbage gets onto the screen.

^R On dumb terminals, those not having the "delete line" function (the vt100 is such a terminal), **vi** saves redrawing the screen when you delete a line by just marking the line with an "@ " at the beginning and blanking the line. If you

* Referring to an upper case letter as a buffer name (A-Z) is the same as referring to the lower case letter, except that text placed in such a buffer is appended to it instead of replacing it.

want to actually get rid of the lines marked with "@" and see what the page looks like, typing a ^R will do this.

.	"Dot" is a particularly useful command. It repeats the last text modifying command. Therefore you can type a command once and then to another place and repeat it by just typing ".".
u	Perhaps the most important command in the editor, u undoes the last command that changed the buffer. Mnemonic: **u**ndo
U	Undo all the text modifying commands performed on the current line since the last time you moved onto it.
[cnt] J	Join the current line and the following line. The <nl> is deleted and the two lines joined, usually with a space between the end of the first line and the beginning of what was the second line. If the first line ended with a "period", then two spaces are inserted. A count joins the next cnt lines. Mnemonic: **J**oin lines
Q	Switch to **ex** editing mode. In this mode **vi** will behave very much like **ed**. The editor in this mode will operate on single lines normally and will not attempt to keep the "window" up to date. Once in this mode it is also possible to switch to the **open** mode of editing. By entering the command **[line number]open<nl>** you enter this mode. It is similar to the normal visual mode except the window is only **one** line long. Mnemonic: **Q**uit visual mode
^]	An abbreviation for a tag command. The cursor should be positioned at the beginning of a word. That word is taken as a tag name, and the tag with that name is found as if it had been typed in a :tag command.
[cnt] ! {motion}{UNIX cmd}<nl>	Any UNIX filter (e.g. command that reads the standard input and outputs something to the standard output) can be sent a section of the current file and have the output of the command replace the original text. Useful examples are programs like **cb**, **sort**, and **nroff**. For instance, using **sort** it would be possible to sort a section of the current file into a new list. Using **!!** means take a line or lines starting at the line the cursor is currently on and pass them to the UNIX command. **NOTE:** To just escape to the shell for one command, use :!{cmd}<nl>, see section 5.
z{cnt}<nl>	This resets the current window size to "cnt" lines and redraws the screen.

4.9. Special Insert Characters

There are some characters that have special meanings during insert modes. They are:

^V	During inserts, typing a ^V allows you to quote control characters into the file. Any character typed after the ^V will be inserted into the file.
[^] ^D or [0] ^D	<^D> without any argument backs up one **shiftwidth**. This is necessary to remove indentation that was inserted by the **autoindent** feature. ^<^D> temporarily removes all the autoindentation, thus placing the cursor at the left margin. On the next line, the previous indent level will be restored. This is useful for putting "labels" at the left margin. 0<^D> says remove all autoindents and stay that way. Thus the cursor moves to the left margin and stays there on successive lines until <tab>'s are typed. As with the <tab>, the <^D> is only effective before any other "non-autoindent" controlling characters are typed. Mnemonic: **D**elete a shiftwidth
^W	If the cursor is sitting on a word, <^W> moves the cursor back to the beginning of the word, thus erasing the word from the insert. Mnemonic: erase **W**ord

<bs> The backspace always serves as an erase during insert modes in addition to your normal "erase" character. To insert a <bs> into your file, use the <^V> to quote it.

5. : Commands

Typing a ":" during command mode causes **vi** to put the cursor at the bottom on the screen in preparation for a command. In the ":" mode, **vi** can be given most **ed** commands. It is also from this mode that you exit from **vi** or switch to different files. All commands of this variety are terminated by a <nl>, <cr>, or <esc>.

:w[!] [file] Causes **vi** to write out the current text to the disk. It is written to the file you are editing unless "file" is supplied. If "file" is supplied, the write is directed to that file instead. If that file already exists, **vi** will not perform the write unless the "!" is supplied indicating you *really* want to destroy the older copy of the file.

:q[!] Causes **vi** to exit. If you have modified the file you are looking at currently and haven't written it out, **vi** will refuse to exit unless the "!" is supplied.

:e[!] [+[cmd]] [file]

Start editing a new file called "file" or start editing the current file over again. The command ":e!" says "ignore the changes I've made to this file and start over from the beginning". It is useful if you really mess up the file. The optional "+" says instead of starting at the beginning, start at the "end", or, if "cmd" is supplied, execute "cmd" first. Useful cases of this are where cmd is "n" (any integer) which starts at line number n, and "/text", which searches for "text" and starts at the line where it is found.

^^ Switch back to the place you were before your last tag command. If your last tag command stayed within the file, ^^ returns to that tag. If you have no recent tag command, it will return to the same place in the previous file that it was showing when you switched to the current file.

:n[!] Start editing the next file in the argument list. Since **vi** can be called with multiple file names, the ":n" command tells it to stop work on the current file and switch to the next file. If the current file was modifies, it has to be written out before the ":n" will work or else the "!" must be supplied, which says discard the changes I made to the current file.

:n[!] file [file file ...]

Replace the current argument list with a new list of files and start editing the first file in this new list.

:r file Read in a copy of "file" on the line after the cursor.

:r !cmd Execute the "cmd" and take its output and put it into the file after the current line.

:!cmd Execute any UNIX shell command.

:ta[!] tag **Vi** looks in the file named **tags** in the current directory. **Tags** is a file of lines in the format:

tag filename **vi**-search-command

If **vi** finds the tag you specified in the **:ta** command, it stops editing the current file if necessary and if the current file is up to date on the disk and switches to the file specified and uses the search pattern specified to find the "tagged" item of interest. This is particularly useful when editing multi-file C programs such as the operating system. There is a program called **ctags** which will generate an appropriate **tags** file for C and f77 programs so that by saying **:ta function<nl>** you will be switched to that function. It could also be useful when editing multi-file documents, though the **tags** file would have to be generated manually.

6. Special Arrangements for Startup

Vi takes the value of **$TERM** and looks up the characteristics of that terminal in the file **/etc/termcap**. If you don't know **vi**'s name for the terminal you are working on, look in **/etc/termcap**.

When **vi** starts, it attempts to read the variable EXINIT from your environment.* If that exists, it takes the values in it as the default values for certain of its internal constants. See the section on "Set Values" for further details. If EXINIT doesn't exist you will get all the normal defaults.

Should you inadvertently hang up the phone while inside **vi**, or should the computer crash, all may not be lost. Upon returning to the system, type:

```
vi – r file
```

This will normally recover the file. If there is more than one temporary file for a specific file name, **vi** recovers the newest one. You can get an older version by recovering the file more than once. The command "vi -r" without a file name gives you the list of files that were saved in the last system crash (but *not* the file just saved when the phone was hung up).

7. Set Commands

Vi has a number of internal variables and switches which can be set to achieve special affects. These options come in three forms, those that are switches, which toggle from off to on and back, those that require a numeric value, and those that require an alphanumeric string value. The toggle options are set by a command of the form:

```
:set option<nl>
```

and turned off with the command:

```
:set nooption<nl>
```

Commands requiring a value are set with a command of the form:

```
:set option=value<nl>
```

To display the value of a specific option type:

```
:set option? <nl>
```

To display only those that you have changed type:

```
:set<nl>
```

and to display the long table of all the settable parameters and their current values type:

```
:set all<nl>
```

Most of the options have a long form and an abbreviation. Both are listed in the following table as well as the normal default value.

To arrange to have values other than the default used every time you enter **vi**, place the appropriate **set** command in EXINIT in your environment, e.g.

* On version 6 systems instead of EXINIT, put the startup commands in the file .exrc in your home directory.

```
        EXINIT='set ai aw terse sh=/bin/csh'
        export EXINIT
```

or

```
        setenv EXINIT 'set ai aw terse sh=/bin/csh'
```

for **sh** and **csh**, respectively. These are usually placed in your .profile or .login. If you are running a system without environments (such as version 6) you can place the set command in the file .exrc in your home directory.

autoindent ai	Default: noai Type: toggle When in autoindent mode, vi helps you indent code by starting each line in the same column as the preceding line. Tabbing to the right with <tab> or <^T> will move this boundary to the right, and it can be moved to the left with <^D>.
autoprint ap	Default: ap Type: toggle Causes the current line to be printed after each ex text modifying command. This is not of much interest in the normal **vi** visual mode.
autowrite aw	Default: noaw type: toggle Autowrite causes an automatic write to be done if there are unsaved changes before certain commands which change files or otherwise interact with the outside world. These commands are :!, :tag, :next, :rewind, ^^, and ^].
beautify bf	Default: nobf Type: toggle Causes all control characters except <tab>, <nl>, and <ff> to be discarded.
directory dir	Default: dir=/tmp Type: string This is the directory in which **vi** puts its temporary file.
errorbells eb	Default: noeb Type: toggle Error messages are preceded by a <bell>.
hardtabs ht	Default: hardtabs=8 Type: numeric This option contains the value of hardware tabs in your terminal, or of software tabs expanded by the Unix system.
ignorecase ic	Default: noic Type: toggle All upper case characters are mapped to lower case in regular expression matching.
lisp	Default: nolisp Type: toggle Autoindent for **lisp** code. The commands ( ) [[and]] are modified appropriately to affect s-expressions and functions.
list	Default: nolist Type: toggle All printed lines have the <tab> and <nl> characters displayed visually.
magic	Default: magic Type: toggle Enable the metacharacters for matching. These include . * < > [**string**] [^**string**] and [<**chr**>-<**chr**>].
number nu	Default: nonu Type: toggle Each line is displayed with its line number.
open	Default: open Type: toggle When set, prevents entering open or visual modes from ex or edit. Not of interest from vi.
optimize opt	Default: opt Type: toggle Basically of use only when using the **ex** capabilities. This option prevents automatic <cr>s from taking place, and speeds up output of indented lines, at the expense of losing typeahead on some versions of UNIX.

paragraphs para	Default: para=IPLPPPQPP bp Type: string Each pair of characters in the string indicate **nroff** macros which are to be treated as the beginning of a paragraph for the { and } commands. The default string is for the **-ms** and **-mm** macros. To indicate one letter **nroff** macros, such as **.P** or **.H**, quote a space in for the second character position. For example: :set paragraphs=P\ bp<nl> would cause **vi** to consider **.P** and **.bp** as paragraph delimiters.
prompt	Default: prompt Type: toggle In **ex** command mode the prompt character **:** will be printed when **ex** is waiting for a command. This is not of interest from vi.
redraw	Default: noredraw Type: toggle On dumb terminals, force the screen to always be up to date, by sending great amounts of output. Useful only at high speeds.
report	Default: report=5 Type: numeric This sets the threshold for the number of lines modified. When more than this number of lines are modified, removed, or yanked, **vi** will report the number of lines changed at the bottom of the screen.
scroll	Default: scroll={1/2 window} Type: numeric This is the number of lines that the screen scrolls up or down when using the <^U> and <^D> commands.
sections	Default: sections=SHNHH HU Type: string Each two character pair of this string specify **nroff** macro names which are to be treated as the beginning of a section by the]] and [[commands. The default string is for the **-ms** and **-mm** macros. To enter one letter **nroff** macros, use a quoted space as the second character. See **paragraphs** for a fuller explanation.
shell sh	Default: sh=from environment SHELL or /bin/sh Type: string This is the name of the **sh** to be used for "escaped" commands.
shiftwidth sw	Default: sw=8 Type: numeric This is the number of spaces that a <^T> or <^D> will move over for indenting, and the amount < and > shift by.
showmatch sm	Default: nosm Type: toggle When a) or } is typed, show the matching (or { by moving the cursor to it for one second if it is on the current screen.
slowopen slow	Default: terminal dependent Type: toggle On terminals that are slow and unintelligent, this option prevents the updating of the screen some of the time to improve speed.
tabstop ts	Default: ts=8 Type: numeric <tab>s are expanded to boundaries that are multiples of this value.
taglength tl	Default: tl=0 Type: numeric If nonzero, tag names are only significant to this many characters.
term	Default: (from environment **TERM**, else dumb) Type: string This is the terminal and controls the visual displays. It cannot be changed when in "visual" mode, you have to Q to command mode, type a set term command, and do "vi." to get back into visual. Or exit vi, fix $TERM, and reenter. The definitions that drive a particular terminal type are found in the file **/etc/termcap**.

terse	Default: terse Type: toggle When set, the error diagnostics are short.
warn	Default: warn Type: toggle The user is warned if she/he tries to escape to the shell without writing out the current changes.
window	Default: window={8 at 600 baud or less, 16 at 1200 baud, and screen size – 1 at 2400 baud or more} Type: numeric This is the number of lines in the window whenever **vi** must redraw an entire screen. It is useful to make this size smaller if you are on a slow line.
w300, w1200, w9600	These set window, but only within the corresponding speed ranges. They are useful in an EXINIT to fine tune window sizes. For example, `set w300=4 w1200=12` causes a 4 lines window at speed up to 600 baud, a 12 line window at 1200 baud, and a full screen (the default) at over 1200 baud.
wrapscan ws	Default: ws Type: toggle Searches will wrap around the end of the file when is option is set. When it is off, the search will terminate when it reaches the end or the beginning of the file.
wrapmargin wm	Default: wm=0 Type: numeric **Vi** will automatically insert a <nl> when it finds a natural break point (usually a <sp> between words) that occurs within "wm" spaces of the right margin. Therefore with "wm=0" the option is off. Setting it to 10 would mean that any time you are within 10 spaces of the right margin **vi** would be looking for a <sp> or <tab> which it could replace with a <nl>. This is convenient for people who forget to look at the screen while they type. (In version 3, wrapmargin behaves more like nroff, in that the boundary specified by the distance from the right edge of the screen is taken as the rightmost edge of the area where a break is allowed, instead of the leftmost edge.)
writeany wa	Default: nowa Type: toggle **Vi** normally makes a number of checks before it writes out a file. This prevents the user from inadvertently destroying a file. When the "writeany" option is enabled, **vi** no longer makes these checks.

SED - A Non-interactive Text Editor

This document is based on a paper by Lee E. McMahon of Bell Laboratories.

Introduction

Sed (a descendent of **ed**) is a non-interactive context editor designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode;
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

Sed is a lineal descendant of the **ED** editor. Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed;* even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ('regular expressions') they recognize; the code for matching patterns is copied almost verbatim from the code for *ed,* and the description of regular expressions in Section 2 is copied almost verbatim from the ROS Reference Manual (9010).

1. Overall Operation

Sed by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line; see Section 1.1 below.

The general format of an editing command is:

```
[address1,address2][function][arguments]
```

One or both addresses may be omitted; the format of addresses is given in Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

1.1. Command-line Flags

Three flags are recognized on the command line:

-n: tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Section 3.3);

-e: tells *sed* to take the next argument as an editing command;

-f: tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, *t* and *b* (see Section 3). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

1.3. Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the *N* command (Section 3.6.).

1.4. Examples

Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

Example:

The command

```
2q
```

will quit after copying the first two lines of the input. The output will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

2. ADDRESSES: Selecting lines for editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces ('{ }')(Sec. 3.6.).

2.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character $ matches the last line of the last input file.

2.2. Context Addresses

A context address is a pattern ('regular expression') enclosed in slashes ('/'). The regular expressions recognized by *sed* are constructed as follows:

1) An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
2) A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.
3) A dollar-sign '$' at the end of a regular expression matches the null character at the end of a line.
4) The characters '\n' match an embedded newline character, but not the newline at the end of the pattern space.
5) A period '.' matches any character except the terminal newline of the pattern space.
6) A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.
7) A string of characters in square brackets '[ ]' matches any character in the string, and no others. If, however, the first character of the string is circumflex '^', the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.
8) A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
9) A regular expression between the sequences '\(' and '\)' is identical in effect to the unadorned regular expression, but has side-effects which are described under the *s* command below and specification 10) immediately below.
10) The expression '\d' means the same string of characters matched by an expression enclosed in '\(' and '\)' earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of '\(' counting from the left. For example, the expression '^\(.*\)\1' matches a line beginning with two repeated occurrences of the same string.
11) The null regular expression standing alone (e.g., '//') is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash '\'.

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the process is repeated.

Two addresses are separated by a comma.

Examples:

/an/	matches lines 1, 3, 4 in our sample text
/an.*an/	matches line 1
/^an/	matches no lines
/./	matches all lines
/\./	matches line 5
/r*an/	matches lines 1,3, 4 (number = zero!)
/\(an\).*\1/	matches line 1

3. FUNCTIONS

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does. The angles around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

3.1. Whole-line Oriented Functions

(2)d -- delete lines

The *d* function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2)n -- next line

The *n* function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

(1)a\
<text> -- append lines

The *a* function causes the argument <text> to be written to the output after the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character ('\') immediately preceding the newline. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

Once an *a* function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. The triggering line may be deleted entirely; <text> will still be written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

(1)i\
<text> -- insert lines

The *i* function behaves identically to the *a* function, except that <text> is written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

(2)c\
<text> -- change lines

The *c* function deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i*, *c* must be followed by a newline hidden by a backslash; and interior new lines in <text> must be hidden by backslashes.

The *c* command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output, *not* one copy per line deleted. As with *a* and *i*, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a *c* function, no further commands are attempted on the corpse.

If text is appended after a line by *a* or *r* functions, and the line is subsequently changed, the text inserted by the *c* function will be placed *before* the text of the *a* or *r* functions. (The *r* function is described in Section 3.4.)

Note: Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Example:

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n         n
i\        c\
XXXX      XXXX
d
```

3.2. Substitute Function

One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The *s* function replaces *part* of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in addresses (see 2.2 above). The only difference between <pattern> and a context address is that the context address must be delimited by slash ('/') characters; <pattern> may be delimited by any character other than space or newline.

By default, only the first string matched by <pattern> is replaced, but see the *g* flag below.

The <replacement> argument begins immediately after the second delimiting character of <pattern>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly *three* instances of the delimiting character.)

The <replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

& is replaced by the string matched by <pattern>

\d (where *d* is a single digit) is replaced by the *d*th substring matched by parts of <pattern> enclosed in '\\(' and '\\)'. If nested substrings occur in <pattern>, the *d*th is determined by counting opening delimiters ('\\(').

As in patterns, special characters may be made literal by preceding them with backslash ('\\').

The <flags> argument may contain the following flags:

g -- substitute <replacement> for all (non-overlapping) instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters; characters put into the line from <replacement> are not rescanned.

p -- print the line if a successful replacement was done. The *p* flag causes the line to be written to the output if and only if a substitution was actually made by the *s* function. Notice that if several *s* functions, each followed by a *p* flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

w <filename> -- write the line to a file if a successful replacement was done. The *w* flag causes lines which are actually substituted by the *s* function to be written to a file named by <filename>. If <filename> exists before *sed* is run, it is overwritten; if not, it is created.

A single space must separate *w* and <filename>.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for *p*.

A maximum of 10 different file names may be mentioned after *w* flags and *w* functions (see below), combined.

Examples:

The following command, applied to our standard input,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubhla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file 'changes':

```
Through caverns measureless by man
Down by a sunless sea.
```

If the nocopy option is in effect, the command:

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

Finally, to illustrate the effect of the *g* flag, the command:

```
/X/s/an/AN/p
```

produces (assuming nocopy mode):

```
In XANadu did Kubhla Khan
```

and the command:

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubhla KhAN
```

3.3. Input-output Functions

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the *p* function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the *w* and <filename>.

A maximum of ten different files may be mentioned in write functions and *w* flags after *s* functions, combined.

(1)r <filename> -- read the contents of a file

The read function reads the contents of <filename>, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If *r* and *a* functions are executed on the same line, the text from the *a* functions and the *r* functions is written to the output in the order that the functions are executed.

Exactly one space must separate the *r* and <filename>. If a file mentioned by a *r* function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

NOTE: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in *w* functions or flags; that number is reduced by one if any *r* functions are present. (Only one read file is open at one time.)

Examples

Assume that the file 'note1' has the following contents:

> Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

Then the following command:

```
/Kubla/r note1
```

produces:

```
In Xanadu did Kubla Khan
        Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson
        and most eminent successor of Genghiz (Chingiz) Khan, and founder of the
        Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

3.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines; they are intended principally to provide pattern matches across lines in the input.

(2)N -- Next line

> The next input line is appended to the current line in the pattern space; the two input lines are separated by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

(2)D -- Delete first part of the pattern space

> Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

> Print up to and including the first newline in the pattern space.

The *P* and *D* functions are equivalent to their lower-case counterparts if there are no imbedded newlines in the pattern space.

3.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)h -- hold pattern space

> The *h* functions copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)H -- Hold pattern space

The *H* function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

(2)g -- get contents of hold area

The *g* function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

(2)G -- Get contents of hold area

The *G* function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2)x -- exchange

The exchange command interchanges the contents of the pattern space and the hold area.

Example

The commands

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

3.6. Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)! -- Don't

The *Don't* command causes the next command (written on the same line), to be applied to all and only those input lines *not* selected by the address part.

(2){ -- Grouping

The grouping command '{' causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the '{' or on the next line.

The group of commands is terminated by a matching '}' standing on a line by itself.

Groups can be nested.

(0):<label> -- place a label

The label function marks a place in the list of editing commands which may be referred to by *b* and *t* functions. The <label> may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A *b* function with no <label> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The *t* function tests whether ***any*** successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

1) reading a new input line, or
2) executing a *t* function.

3.7. Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The *q* function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

Ridge Computers
Corporate Headquarters

2451 Mission College Blvd.
Santa Clara, California 95054
Phone: (408) 986-8500
Telex: 176956