CONVEX Text Editor
User's Guide

Document No. 740-000430-000

Version 1.0
March 8, 1985

CONVEX Computer Corporation

Table of Contents

List of Tables

Preface

Introduction

UNIX is a sophisticated operating system developed for 16-bit minicomputers at Bell Labs in the early Seventies. After adaptation to 32-bit machines, UNIX was significantly enhanced by a group of software designers at the University of California at Berkeley. The CONVEX implementation of UNIX is a modified version of Berkeley UNIX, UNIX 4.2BSD.

Five text editors initially developed for use with UNIX are included in the CONVEX-1 software distribution. There is also an optional editor, *Emacs*, which is available through your CONVEX sales representative. This manual documents the use of three of editors--*ex, sed,* and *vi. Ex* is a line editor which allows you to manipulate text one line at a time. *Sed* is a stream editor used for the non-interactive batch processing of editing scripts. *Vi* is a screen editor that can display and manipulate a screenful of text at a time. The *CONVEX UNIX Tutorial Papers* contains information on how to use the *ed* and *edit* editors. The *Emacs User's Guide* documents the use of *Emacs.*

Objectives and Intended Audience

The purpose of this document is to provide you with step-by-step instructions on how to use the *ex, sed,* and *vi* editors. The document addresses all users of the CONVEX-1.

Organization

The document is organized as follows:

- Chapter 1 provides an introduction to text editing and describes how editors function. Additional topics addressed include how to choose an editor and the capabilities of the various text-editing packages.
- Chapter 2 explains the basic use of *vi* and references additional documentation.
- Chapter 3 explains the basic use of *ex* and references additional documentation.
- Chapter 4 describes the use of *sed* and references additional documentation.

Notational Conventions

The following conventions have been used in this document:

- ASCII nonprintable characters are designated by their mnemonic enclosed in 'less than' and 'greater than' signs. For example, <CR> stands for "carriage return."
- **Boldface** type indicates user-entered information for a computer program.
- Brackets [ ] designate optional entries.
- A horizontal ellipsis ... shows repetition of the preceding item(s).
- A vertical ellipsis shows continuation of a sequence where not all of the statements in an example are shown.
- *Italics* indicate program names, commands, or filenames.
- References to the *CONVEX UNIX Programmer's Manual* appear in the form *vi*(1), where the name of the manual page is followed by its section number enclosed in parentheses.
- The up arrow ↑ designates the control key. When ↑ appears before another character, it means to press the indicated key while holding down the control key.
- Uppercase and lowercase letters many times represent different commands. Be sure to type commands in the case in which they appear in the examples.

Associated Documents

Users are urged to consult the following documents, available from CONVEX Computer Corporation. Ordering information and mailing lists are included at the back of this manual.

- *CONVEX UNIX Programmer's Manual, Parts I and II* (710-000150-100 and 710-000151-100) contains complete reference material on the operating system for CONVEX computers.
- *CONVEX UNIX Tutorial Papers* (710-000250-100) is a collection of previously published papers that provides instruction in document preparation, programming, text editing, supporting tools and languages, system maintenance, and system implementation.
- *UNIX Primer Plus* (710-000121-000) M. Waite, D. Martin, and S. Prata; Indianapolis, Indiana; Howard W. Sams & Co., Inc., 1983. This book is an introduction to the UNIX operating system.
- *Emacs User's Guide* (740-000021-000) documents the use of the *Emacs* text editor.

1

An Introduction to Computer Editing Systems

Overview

This chapter discusses the capabilities of text editors in general. It is intended for the those who have little or no experience with text editors. This chapter includes:

- A description of basic text editing functions
- A functional comparison of line, screen, and stream editors
- Recommendations on the type of text editor to use for various applications
- An overview of the CONVEX UNIX editors: *ed, edit, ex, vi,* and *sed*
- An explanation of basic terms, concepts, and operations

What is a Text Editor?

A text editor is a computer program that enables you to create, modify, and display text by typing commands on a terminal. The text editor is indispensable since you create every line of code or text using it.

Most text editors enable you to type in text, correct typing mistakes, edit copy or programs, and display parts of a program or document. Text editors enable you to search a file for terms or phrases, insert text from other files, build tables, and pass commands to a command interpreter.

Types of Text Editors

Generally speaking, there are three types of text editors: line editors, screen editors, and stream (or batch) editors. *Line editors* display specific file lines, but only upon your request. Line editors require you to specify both a command and the line number or numbers of the text to be modified or moved.

Screen editors display text a "screenful" at a time. The screen is continually updated to mirror the latest change in the file. Screen editors require that you place the cursor on the text to be modified prior to entering a command.

Stream editors are noninteractive batch-mode editors. These editors are primarily used for making global changes to large files. Screen editors are best for general-purpose editing.

Which Type of Editor Should I Use?

The only type of editor that can be used on hardcopy devices and certain types of terminals is the line editor. It operates rapidly, uses relatively few machine cycles, and is easy to learn. Despite these advantages, however, most users with access to a CRT prefer using screen editors such as *vi*

or *emacs*. Screen editors offer the same capabilities as line editors, with several added conveniences. Handling large files becomes much easier, since screen editors provide "scrolling" capabilities which enable you to page through a file rather than engage in the cumbersome business of specifying line numbers. Making changes within a line is easier, since words can be retyped within a given line. In addition, screen editors always display a full page of text. The screen editor is typically more complicated to use than the line editor, but the added convenience is worth the effort.

For general purpose editing, you will do best to use either *vi* or *emacs*. *Vi* operates in two modes: command and insert. While in insert mode, the characters you type are inserted into the file and displayed on the screen. While in command mode, you enter commands on the command line to perform cursor movement and other editing functions. *Emacs* operates only in insert mode and has many escape sequences for performing various editing functions. You may want to compare these two editors and select the one with which you feel more comfortable. *Emacs* is an optional editor available through your CONVEX representative.

Stream editors, as mentioned previously, are best used to make global changes to large files. They are not well suited for everyday editing, but are useful tools when you don't want to make massive changes with an interactive editor.

CONVEX UNIX Text Editors

Ed was the first UNIX text editor developed. *Ed* is very simple to learn and to use, but is of limited usefulness owing to its restricted command set. Brian Kernighan's "A Tutorial Introduction to the UNIX Text Editor" is an excellent resource for those interested in learning to use *ed* (see the *CONVEX UNIX Tutorial Papers).*

Ex is the root of a family of editors: *edit, ex,* and *vi. Ex* was developed at Berkeley and is based on *ed. Ex* offers many of the same capabilities as *ed,* but with an expanded, more sophisticated command set.

Edit is a friendlier version of *ex. Edit* has a smaller command set, and although less powerful than *ex,* is simpler to use and to operate. If you decide to use a line editor, *edit* may be a good one to start with. *Edit* is documented in "Edit: A Tutorial," by Ricki Blau and James Joyce (see the *CONVEX UNIX Tutorial Papers).*

Sed is a stream editor which processes sets of commands in batch mode. Unlike most editors, which match an interactive command to a set of input lines, *sed* accepts a set of commands and then works through a file line by line, changing those lines which are affected by the input commands. *Sed* is useful for making simple changes to very large files and for batch-mode editing within shell scripts.

Vi is the screen-oriented version of *ex.* The *vi* command set is a superset of the *ex* command set; in fact, you can access the *ex* command set from within *vi.*

Some Useful Definitions

Successful text editing depends on your ability to manipulate both a command set and a set of concepts unique to text editing. The most useful concepts are defined in this section. You should become familiar with this list before moving on to subsequent chapters.

- *File:* In the simplest sense, a file is a collection of bytes. More importantly, a file is the logical entity constituting a unit of text or a program. User files are created when an editor writes out the current "best buffer".

- *Text Buffer:* Accessing a file with a text editor copies the file into a "buffer". All modifications made to a file during an editing session are made to the copied file in the

buffer. This attribute protects the file from inadvertent or careless mistakes. When you complete the editing session and want to save the contents, "write" the buffer over the file, which will erase the original contents. No changes are made to the file until the buffer is written.

- *Line:* A line is a sequence of zero or more characters followed by a <CR>.

- *Line Number:* When considering a file as a collection of lines, the lines are numbered consecutively with line one being the first line in the file. A line's number is used to reference it.

- *Current Line:* The current line is the last line affected by a command. You must know the current line number in order to perform many editing operations successfully. Fortunately, in most cases, the cursor is positioned at the current line. In some instances, you must undertake special operations to determine which line is the current line. These operations are discussed in the chapters on *sed* and *vi*.

2

Using Vi

Introduction to Vi

This chapter discusses the basic capabilities of the *vi* text editor. *Vi* is a display-oriented version of *ex,* the text editor discussed in Chapter 3. The entire *ex* command set is available within *vi.* In addition, there are commands for display-editing operations. This document addresses only those commands which are most commonly used. If you are interested in expanding your level of competence, read William Joy's "An Introduction to Display Editing with Vi" in the *CONVEX UNIX Tutorial Papers.*

This chapter describes

- Using the special characters <ESC>, <CR>, and <DEL>.
- The *vi* command set and command structure.
- How to invoke the editor.
- How to enter new text.
- Using the cursor and the editing window.
- Editing operations instructions for exiting the editor.

Special Characters

The escape <ESC>, return <CR>, and interrupt keys all have very important uses within *vi.* (The interrupt key is generally labelled DEL or DELETE. If you are not sure which key is used to signal an interrupt, contact your system manager.)

The <ESC> key has three functions: it signals the end of a text stream you are inserting into a file, it cancels the execution of a partially typed command, and serves as a "security blanket." If you become confused while in the editor, press <ESC> until the bell on your terminal rings. The beep signals that *vi* is now listening to you and is awaiting a command.

<CR> is used to terminate many commands. The interrupt key is used to send an interrupt to the editor; in essence, the interrupt key forces the editor to listen to you when you do not like what is going on.

Command Structure

Since each editing operation must be explicitly defined, the *vi* command set is very large. Luckily, mnemonics are used in place of command names in nearly all instances. For a complete list of the *vi* commands, see "An Introduction to Display Editing with Vi" in the *CONVEX UNIX Tutorial Papers.* If you mistakenly type a key sequence that does not represent a *vi* command, your terminal will beep at you.

The basic structure for all *vi* commands is

[count] operator [count] object

where *operator* is the command sequence and *object* is the entity the operator affects. (The object may be another command sequence.) The optional *count* is used as multiplicative operator. For example:

dw

causes the next word to be deleted. The character *d* is the operator, while the character *w* is the object. The commands:

3dw
d3w

both have the same effect--that of deleting the next three words.

4d3w

causes the next 12 words to be deleted. Remember that typing a colon enables you to execute *ex* commands from within *vi*.

Invoking the Editor

Invoking *vi* is a very simple operation. For example, to access a file named *chaucer*, simply type

vi chaucer

If the file exists, it will be copied into an editing buffer. *Vi* does not directly modify files. Changes are made to the copy which exists in the editing buffer. The original file is not changed unless and until you "write", or copy, the contents of the buffer back into the original file.

When the file to be edited has been copied into the buffer, the terminal screen will be cleared, and the cursor will be set to the first character of the first line of the editing window. A one-line listing of the name of the file, along with its length in lines and characters, will be displayed at the bottom of the window. If the file to be edited is new (i.e., nonexistent), the cursor will appear in the home position (leftmost position of the first line). A one-line message containing the name of the file and the text [New File] will be displayed at the bottom of the screen.

Vi has two modes: command mode and insert mode. Command mode is the initial mode you will be in after you invoke the editor. It is used to enter editing commands. Insert mode, as the name implies, is used to insert lines of text. To switch from command to insert mode, type the character *i*. When you finish inserting text, press <ESC> to return to command mode. If you become confused about which mode you are in, type <ESC> until the terminal bell beeps. The beep indicates that *vi* is now in command mode.

Cursor and Window Operations

Moving the Cursor Within the Editing Window

You can move the cursor in several different ways. The arrow keys move the cursor up, down, or to the left or the right, if your terminal keyboard is so configured. Alternatively (and preferably for touch typists), you can move the cursor left with the *h* key, right with the *l* key, down with *j*, and up with *k*. To move the cursor more than one character at a time, press the key until the cursor has moved as far as you would like. You can achieve the same effect by specifying a count before pressing the appropriate key. For example, typing *15l* will move the cursor to the right 15 spaces.

The + key is used to move the cursor to the next character which begins a line. The - key functions like the + key, but in the opposite direction.

To scroll upward and downward on the screen, use the ↑*D* and ↑*U* command sequences, respectively. ↑*E* will expose another line at the bottom of the file. ↑*Y* exposes a line at the top of the file.

To page forward and backward, use the ↑*F* and ↑*B* command sequences, respectively. ↑*H* backspaces over individual characters (like *h* does), while ↑*W* backspaces over and erases an entire word.

Other Cursor Movement Operations

The following are commands to move the cursor forward or backward in various increments. To move the cursor with these commands, position the cursor to the line containing the text to be modified and type the command character. With these commands, the cursor will continue to move to the next line when it reaches the end of a line.

w	Moves the cursor to the next word in the line.
b	Moves the cursor back to the previous word in the line.
e	Moves the cursor to the end of the current word.
W	Moves the cursor to the next word in the line without stopping for punctuation marks.
B	Moves the cursor to the previous word in the line without stopping for punctuation marks.
(	Moves the cursor to the beginning of the previous sentence. You can specify a repetition count to move the cursor over a number of sentences. For example, 5(moves the cursor back 5 sentences.
)	Moves the cursor to the beginning of the next sentence. You can specify a repetition count to move the cursor over a number of sentences. For example, 5) moves the cursor forward 5 sentences.
}	Moves the cursor forward a paragraph. *Vi* recognizes paragraphs as either beginning with the first line after a blank line, or as beginning where defined by *nroff* macros.

[[	Moves the cursor forward a section in *nroff* text. *Nroff* is documented in the *CONVEX UNIX Tutorial Papers*. If text is C source code, this command moves the cursor over an entire procedure.
]]	Moves the cursor backward a section in *nroff* text. If text is C source code, the cursor moves over an entire procedure.
↑	Moves the cursor to the first nonblank character on the current line.
0	Moves the cursor to the end of the current line.
$	Moves the cursor to the end of the current line.
<CR>	Moves the cursor to the beginning of the next line.
-	Moves the cursor to the beginning of the previous line.
+	Moves the cursor to the beginning of the next line.
H	Moves the cursor to the home position (leftmost position in the topmost line of the screen). The screen does not scroll.
M	Moves the cursor to the middle of the screen.
L	Moves the cursor to the last line of the screen.

Finding Specific Characters or Columns

The editor can be instructed to search a line for the next occurrence of a single character with the *f* command. For example, typing *fn* causes the cursor to move to the next occurrence of the character "n". Having found the first "n", you may want to continue the search. To do so, type a *;* character. The cursor will move forward to the next occurrence of "n" each time you type *;*. To return to one of the characters previously located, type a , character. The cursor will back up to the previous occurrence of the character each time you type ,.

F is used like *f*, but searches the line backwards for the specified character. The *;* character is used again here to continue the search for the specified character. As before, use the , character to redirect the search in the opposite direction. Table 2-1 summarizes these operations.

Table 2-1: Search Commands

Command	Result
fx	Search forward for character x
Fx	Search backward for character x
;	Search for same character in same direction
,	Search for same character in opposite direction

Any of these commands can be given leading counts. For example, typing *6fn* will move the cursor to the sixth occurrence of "n".

Two other cursor-movement commands are *t* and *T*. *T* causes the same effect as *F*, but positions the cursor directly to the right of the specified character. The *t* command works like *f*, but positions the cursor just to left of the specified character. These commands are used most frequently in conjunction with delete and change operations.

A final method of positioning the cursor is to use the | command. This command moves the cursor to a column number specified in a prepended count. For example, the command *40|* moves the cursor to column number 40. The procedures used to locate specific lines and text strings are described later in the chapter.

Window Operations

Vi displays 23 lines in the editing window. If you decide to change the size of the window, use the *set* command. For example, to set the window size to 10 lines, type

:set window=10 <CR>

To replot the screen, type the command sequence *^L*.

Particular lines can be placed at the top, bottom, and middle of the window by using the various permutations of the *z* command. To locate a given line in one of these positions, first position the cursor on the line to be moved and then enter the *z* command. To place the line at the top of the window, follow the *z* command with a <CR>. Following the *z* command with a '.' character will move the line to the center of the screen, and appending a '-' to the character will cause the line to be positioned at the bottom of the screen.

Entering New Text

To insert text into a file, use one of the following commands. Any of these commands puts you in insert mode; while in insert mode, anything you type will be entered into the file as text. The commands used to enter insert mode are:

a Appends text after the current cursor position. Press <ESC> to return to command mode.

A Appends text at the end of a line. This command inserts text at the end of the line, regardless of where the cursor is positioned on that line. Press <ESC> to return to command mode.

i Inserts text before the current cursor position. Press <ESC> to return to command mode.

I Inserts text before the first nonblank position of a line. This command inserts text at the first nonblank position of the line, regardless of where the cursor is positioned on that line. Press <ESC> to return to command mode.

o Opens new lines below the current line enabling you to insert text. Press <ESC> to return to command mode.

O Opens new lines above the current line enabling you to insert text. Press <ESC> to return to command mode.

For all of these commands, the insert can run onto multiple lines by using <CR> within the insert. Specifying a count causes the inserted text to be replicated, but only if the inserted text is all on one line.

Most typing errors can be corrected by backspacing. If you make a more serious error, type <ESC> to stop the text addition, move the cursor to the appropriate spot and correct the error using one of the commands described in subsequent sections. When the error has been corrected, use *a* or *A* (depending on the cursor position) to continue to add text.

If the text input is lengthy, it is a very good idea to save or write text periodically by using the *w* command. Writing the text with the *w* command destroys the file's old contents. The *ZZ* command is used to write the file's contents and quit. Other methods of quitting the editor are discussed later in this chapter.

Making Text Changes

Changing Words and Sentences

There are three commands for changing words and sentences: *cw, c)* and *c(.*

The *cw* command changes the next word after the current location of the cursor. The word (or words) are replaced with the text following the command. <ESC> signals the end of the replacement text. For example, to change a word to "hello," type

cwhello <ESC>

If the change modifies less than one complete line, then the last character to be changed is marked with a $ character.

Change sentences with the *c)* and *c(* commands. The *c)* command is used to change the current sentence. The *c(* command is used in either of two ways, depending on the cursor position. If the cursor is positioned at the beginning of a sentence, the command changes the previous sentence. If the cursor is not positioned at the beginning of a sentence, the command changes the part of the sentence which precedes the cursor.

In the case of any of these commands, you can prepend a repetition count to the command. All of these commands end with <ESC>.

Deleting Text

There are *vi* commands for deleting text a character at a time; deleting text in units of words, sentences, and paragraphs; and deleting parts of lines.

Two commands are used to delete text a character at a time. The *x* command deletes characters moving forward, and the *X* command deletes characters backward along the line. You can give either of these commands a prepended count. For example, *3x* deletes the next 3 characters, while *3X* deletes the previous 3 characters. Neither command will delete characters beyond the boundaries of the current line.

If you repeat the *x* command after the last character on the line is deleted, *x* will change direction and delete characters backward along the line. This phenomenon is fairly entertaining and saves keystrokes in certain situations.

To delete words, place the cursor on the first character of the word to be deleted and type *dw* (delete from here to end of word). You can prepend a count to either the operator or the object to delete multiple words. *4dw,* for example, deletes the next 4 words just as *d4w* does. Another way to do this is to use the '.' command, which repeats the last command that changed the buffer. The effects of the '.' command appear at the current cursor location. (You must have moved the cursor since the last command.) Specifying *dw...* deletes the next four words. The editor automatically adjusts the spacing.

You delete sentences in much the same way as words. Place the cursor at the beginning of the sentence to be deleted and type the *d)* command. To delete consecutive sentences, prepend a count character to the command. For example, *4d)* and *4)d* will both delete the next four sentences. So will the *)d...* command. The editor will join the sentences on either side of the

sentences which have been removed.

Commands used for deleting paragraphs, and *nroff* and C sections, use the same symbols as the cursor movement commands. The *d}* and *d{* commands delete preceding and following paragraphs, respectively. Delete *Nroff* and C sections with the *d]]* and the *d[[* commands.

Replacing and Transposing Characters

Four commands are used for character replacement operations: *r, R, s* and ~.

The *r* command is used to replace a single character. To use the command, first position the cursor on top of the character to be replaced. Next, type the character *r* and the replacement character. For example, the command *r1* replaces the character marked by the cursor with the character *1*.

The *R* command replaces characters on a given line with the characters specified in the command. For example, entering

Rglitch <ESC>

replaces six characters with the string *glitch*. Replacement begins at the current location of the cursor.

The *s* command replaces the character marked by the cursor with all of the text entered before an <ESC>. You may use a count to change more than one character. For example, to change the text string "zz" to "abcd", enter the following command sequence:

2sabcd <ESC>

When you use a count with the *s* command, the last character to be changed is marked with a $ symbol.

The tilde character ~ is used to toggle the case of the character under the cursor. The cursor moves to the right one character after the editor makes the replacement.

The easiest way to make a correction requiring the transposition of characters is to place the cursor over the first of the misplaced characters, then to type *xp*. *x* deletes the first character, and leaves the cursor sitting on the second character. The deleted text moves to an unnamed buffer. The *p* command puts the deleted character back in the word after the second character.

Moving Blocks of Text

Moving blocks of text of any size is accomplished by using the *delete* and *put* commands. The text to be moved is first deleted using the method previously discussed. The *p* (*put*) command is then used to put the deleted text back after the current line. *P* places the deleted text before/above the cursor.

The text is placed as whole lines if whole lines were deleted. Otherwise, the text is inserted between the characters before and after the cursor. For example, if you wanted to move the second and third lines of the following file:

When that April with his showers sweet
Has pierced the drought of March to the root
And bathed every vein in such liqour
As has virtue to engender the flower;
When, also, Zephyrus with his sweet breath
Has inspired in every wood and heath

you would first move the cursor to the beginning of the second line. Next, enter the *delete* command:

2dd

Note that the command *dddd* cannot be substituted for this operation as only the last operation is stored in the unnamed buffer; prior operations are stored in other numbered buffers. Finally, move the cursor to the end of the file and enter the *put* command:

p

The *p* command can also be used to recover up to the last nine items deleted. The command *"1p* for example, recovers and inserts the most recently deleted text. *"2p* causes the text deleted with the next-to-last delete operation to be restored, and so on.

Copying Blocks of Text

To copy blocks of text from one location to another, use the *yank* and *put* commands. The operation is similar to the one used for moving text. *Yank* (*y*) is an operator that pulls the following object into a temporary buffer. As was noted before, *put* is used to retrieve the text from the buffer. *Y* is used to yank a copy of the current line into a temporary buffer. So, for example, to copy the first line of a file three times, move the cursor to the line to be copied and enter *Y*. Now move the cursor to the desired location and enter the *put* command three times.

Both of these commands can be used with a prepended count. Named buffers (described later) can also be used with either command.

Locating Lines and Text Strings

To locate a particular line by number, use the *G* (goto) command with the line number to be located prepended to the command. For example, the command sequence *200G* positions the cursor at line 200. The file window will change as necessary to keep the line specified positioned in the middle of the screen.

You can also position the cursor in the file by instructing the editor to look for a particular text string in the file. To accomplish this operation, type the / character followed by a string of characters and, lastly, by <CR>. The editor will position the cursor at the first occurrence of this string. Locate the next occurrence of the string by typing *n*. The character ? instructs the editor to search backwards from the current cursor position.

The *N* command looks for the next occurrence of the string in the direction opposite to that of the original search, while *n* continues the search in the same direction.

If the search string given to the editor cannot be located, the editor will print a one-line error message on the last line of the screen, and the cursor will return to its initial position.

The ↑*G* command is used to inquire about the state of the current file. The editor will show you the name of the file you are editing, the current line number, the number of lines in the buffer, and the percentage of the buffer after the cursor. If you somehow become lost in your file, typing the double quote character " will return you to your previous position.

The *m* (mark) command can be used to mark a position within a file. You can distinguish different marks by adding a single character after the command. For example, *ma* marks the line containing the cursor with the label *a*. To return to that particular position from any position simply type the command *'a*.

Using the Undo Commands

Unlike many other environments, *vi* offers an *undo* command for resolving careless actions. There are two versions of the command.

If you mistakenly enter a command whose effects you deplore, simply type the character *u*. The last change made to the file will be reversed. As you will note with continued usage, a second *undo* reverses the action of the first, reinstating the effects of the original command. Note that *undo* only reverses the last change made; you may not back up and undo previous commands.

A second version of *undo* is the *U* command. *U* restores the current line to its original state, no matter how many changes have been made to it. ("Original state" refers to the configuration of the line before *any* editing changes made during the current edit of the line.) Unfortunately, *U* will not undo itself.

Using Named Buffers

Unnamed buffers are easy to use since you need not specify a name and the contents disappear automatically after the editing session. However, there are two disadvantages to using unnamed buffers. The unnamed buffer always contains the last text deleted, changed, or yanked. Since you may access only the last text to be modified, you must complete move and copy operations immediately after the deletion or yank has occurred. Furthermore, you cannot transfer text between files since calling up the second file automatically clears the unnamed buffer.

Both of these disadvantages can be avoided by using named buffers. There are 26 named buffers, with names ranging from a to z.

Named buffers are used just as unnamed buffers are, except that they must be explicitly referenced in commands. For example, the command *3dd* deletes three lines and places them in an unnamed buffer. To place the lines in a named buffer, say buffer j, enter *"j3dd*. The same three lines can be retrieved for a *put* operation with the command *jp*.

Named buffers are referenced with the command *"x*, where *x* is the named buffer to be referenced.

Unlike unnamed buffers, named buffers are preserved when new files are called up for editing. You can easily transfer text to other files simply by referencing the named buffer with a *put* command after the new file has been moved into the editing buffer.

Using Ex Commands

You can use any of the *ex* commands (see Chapter 3) from within *vi*. To issue an *ex* command from within *vi*, type a colon *:* while in *vi* command mode. The cursor will move to the bottom line of the screen. You can then enter an *ex* command.

Quitting the Editor

A variety of mechanisms exist for exiting the editor. The *ZZ* command writes the contents of the editing buffer back into the file being edited before the editing session is terminated. The *:w* works the same as *ZZ*. Entering

:q!<CR>

signals the editor to terminate the session *without* saving any of the modifications made to the editing buffer.

3

Using Ex

This chapter includes information on invoking, using, and exiting from *ex*. The information included is tutorial in nature and is intended to introduce the new user to the basics of using *ex*. For additional information, please refer to the "Ex Reference Manual (Version 3.5/2.13--September, 1980)" by William Joy and Mark Horton (see Section 2 of the *CONVEX UNIX Tutorial Papers*).

Introduction

As mentioned previously, *ex* is the root of a family of editors which includes *edit* and *vi*. *Ex* was developed at Berkeley in the late Seventies, and is the most highly evolved line editor available on the CONVEX UNIX system. *Ex* offers both an expanded command set and procedures for recovering files after a system crash. *Ex* presents fewer limitations on file size than *ed*.

Invoking the Ex Editor

To access *ex* , type the following command on the command line:

ex filename

where *filename* is the name of the file to be created or accessed. If you request a nonexistent file, *ex* creates a new file and displays a "New File" message.

Once invoked, the editor will respond with a one-line listing containing the name of the file to be edited and its length in terms of lines and characters. The *ex* prompt character : will appear at the beginning of the next line.

At this point, you can begin entering *ex* commands. Remember, however, that you are really manipulating text in a buffer. Until you write the file with the *w* command, none of the changes you make will be permanently recorded in the file.

Ex can be invoked with more than one filename, as follows:

ex file1 file2 file3
3 files to edit
"file1" 400 lines, 20999 characters

As the example above indicates, *ex* responds with a one-line listing containing the number of files to be edited. After displaying this listing, *ex* reads into the editing buffer the first file named. *Ex* then displays a one-line listing containing the length in words and characters of the file to be edited. The next file named on the command line is accessed with the *n* command. For example, in the above example, if you are finished with *file1* and want to edit *file2*, type:

:n file2

Ex Command-Line Options

The command formats shown in the previous section illustrate basic, frequently used methods of invoking the editor. There are, however, other ways to enter the editor, which depend on the use of various command-line options. This section describes the more commonly used command-line options. The following prototype illustrates their use:

ex [-] [-r] [-R] [+command...] name...

The - option shown here supresses all of the feedback generally supplied by the editor (error messages, prompts, etc.). This option is typically used to process editor scripts in command files. Use the *-r* option to recover file modifications after a system crash. The *-R* option sets the *readonly* option used to look at files which you do not plan to modify. The *+command* argument causes the editor to begin execution by processing the specified command; otherwise, the editor is positioned at the last line of the first file specified on the command line. Specify multiple filenames by including them in sequence at the end of the command line (as the *name...* argument indicates.)

Remember that the command-line options described here are a subset of those available. Consult the "Ex Reference Manual" in the *CONVEX UNIX Tutorial Papers* if you require more extensive documentation.

Setting the Editor Profile

More than 40 options are included in the *ex* distribution. As opposed to the commands, which direct the editor to undertake specific actions on a file for a period, options modify the operation of the editor itself. Turned on options using the *set* command; generally, you would use an abbreviated form of the option. For example, to use the *number* command, which causes line numbers to be displayed with output lines, type:

:set nu

Alternatively, you can write default option settings into the *.exrc* file in your home directory. (The use of this file is briefly described in the first section of the "Ex Reference Manual." The manual also includes a listing of all the options available in *ex,* with their abbreviations and defaults.) Table 2-1 briefly describes the more commonly used options.

Unset or turn off options by typing the letters *no* before the option name. For instance, to turn off the *errorbells* option, which causes the terminal bell to ring when an error message is displayed, type:

:set noeb

If you use an argument with an option, the argument follows the option name, separated by an equal sign. To set the scrolling distance to 22 lines, for example, type:

:set scroll=22

You can set or unset one or more options with the same *set* command by separating the options with spaces. For example, to accomplish all of the operations shown as examples so far, type:

:set nu noeb scroll=22

You can also use *set* to display current option default settings. Display individual option settings by typing a question mark after the option name, as follows:

:set scroll ?

The *set all* command displays a listing of the settings of all the options.

Table 2-1 provides an introduction to the use and format of the most commonly used *ex* options. Recall that options preceded by the letters *no* are inoperative.

Table 3-1: Ex Options

Option	Function	Default
autoprint	Prints current line after each command	autoprint
autoindent	Aligns cursor indentation	noautoindent
autowrite	Buffer contents are written to current file after modification or under unusual circumstance	noautowrite
beautify	Strips unimportant control characters from input lines	nobeautify
number	Output lines are printed with line numbers	nonumber
terse	Shortens error messages	noterse
report	To help prevent catastrophes, *ex* prints a warning when a large number of lines are changed by a command. *Report* changes the default definition of "large" from five to a number that you select.	report=5
showmatch	Cursor moves to matching '(' or '{' for one second before returning to initial ')' or '}'	noshowmatch

Basic Editing Operations

This section describes how to:

1. Add text to the file
2. Display text
3. Change or delete lines of text
4. Replace character strings

5. Use the *undo* command
6. Read new files into the editing buffer
7. Write the editing buffer to a file
8. Quit the editor

Adding Text to the File

To add text to a file, use either of two commands--the *a* command adds text to an empty file or after the specified line in an existing file. The *i* command also inserts additional text into an existing file. (Note that the *i* command inserts text before the current line.) In each case, type the command on the command line (after the colon). The text to be added or inserted is added on a new line. Terminate text input with a period typed by itself on a new line. For example:

```
:a
When that April with his showers sweet
Has pierced the drought of March to the root
And bathed every line in such liqour
As has virtue to engender the flower
.
:
```

Using the "set autoindent" (*set ai*) option simplifies the typing of programs. *ai* automatically aligns the cursor at the same level of indentation as the previous line. This depth of indentation continues until either you cancel the indentation, or until you change the indentation by putting tabs or spaces at the beginning of the next line. Typing ↑D causes the cursor to backspace over the indentation.

Displaying Lines of Text

If the *autoprint* option has been set (via a "set autoprint"), there is little need for a *print* command, since lines are automatically displayed each time you reference or change them. For example:

```
: 1,2
When that April with his showers sweet
Has pierced the drought of march to the root
```

If you decide not to use the *autoprint* option, you will have to use the *p* command to display lines of text.

The *l* command displays control characters along with text. Control characters, which modify data, are usually never printed. Hence, their inadvertent or misplaced inclusion in a text file can cause serious (and undetectable) problems in a text file. The *l* command prints the control characters with a preceding ^ character, so that you can identify the characters.

The *#* command causes line numbers to be displayed. Obviously, setting the *number* option renders this command redundant.

The *z* command displays a screenful of text underneath the current line. The *z=* command causes the current line to be marked:

```
: 4z=
When that April with his showers sweet
----------
Has pierced the drought of March to the root
----------
And bathed every vein in such liquor
As has virtue to engender the flower
```

In each case, the last line displayed becomes the new current line. In this case, the current line remains the one marked on the screen.

Changing or Deleting Text

The commands for changing and deleting text are *c* and *d*, respectively. Typically, you change text by specifying an initial line address and the number of lines to be changed. For example:

```
: 1c2
When, also, Zephyrus with his sweet breath
Has inspired in every wood and heath
:
```

replaces lines 1 and 2 of the current file with the input text. The last line input becomes the current line.

You delete text in much the same way. For instance, to delete 10 lines of a given file, beginning with line 7, type:

```
: 7d10
```

The line after the last line deleted becomes the current line; if the lines were originally at the end, the new last line becomes the current line. If you specify a named buffer (a through z), the specified lines are saved in that buffer. If you specify a named buffer using an uppercase letter, the lines are appended to that buffer rather than replacing the contents. (See the Section "Advanced Editing" later in this chapter for examples using named buffers.)

Substitution Operations

The *s* command performs substitution operations. The format of the command is:

:line1, line2 s /pat/repl/ options count flags

where *pat* is the pattern to be replaced and *repl* is the replacement text. The first instance of *pat* is replaced with *repl* on each line from *line1* to *line2*, inclusively.

An example of basic substitution follows.

```
:1 s/that/hat
: When hat April with his showers sweet
```

If you specify the global indicator option *g* , then substitution is made throughout the line. Otherwise, only the first occurrence of the string in the specified line is changed. The following example replaces every occurrence of the string "that" with "When":

```
:s/that/When/g
```

The *c*, or confirmation, flag is very useful and can be used with the substitution command. When this flag is used, substitution does not occur automatically. Instead, before each substitution, the line to be substituted is marked with ^^^ characters. Typing the character *y* effects the substitution; any other input causes no change. An example of this operation follows:

```
: g/eath/s//eeze/c
When, also, Zephyrus with his sweet breath
                                ^^^^
:y
Has inspired in every wood and heath
                              ^^^^
```

In this example, *ex* looked for the first occurrence of the string "eath," and then waited for a signal before changing the string. Since *y* was specified, "breath" was changed to "breeze". Since no signal was transmitted in the second case, though, the word "heath" was left unchanged.

The Undo Operation

Undo negates any changes made in the editing buffer by the last editing command. *Undo* also repositions the current line to the position it was in when the last editing command was issued. Global commands are completely undone by *undo;* that is, every change made by a global command is negated by *undo. Undo* also acts as its own inverse. A second *undo* will cause the changes previously undone to be reinstated.

Reading New Files into the Editing Buffer

Text from other files can be inserted into the editing buffer with the *read* (*r*) command. Specify a line number and a filename with the command as follows:

```
: 17r foo
: 1r chaucer
```

The file named is read into the buffer beginning on the line after the line specified on the command line. If you do not specify a line number, the file is placed in the buffer after the current line. The last line read into the buffer becomes the new current line.

You can select another file for editing with the *e* command. If you have not written out the contents of the first edit buffer, however, *ex* displays the following warning:

```
: e another
No write since last change (edit! overrides)
:
```

As the message indicates, the ! variant of the *edit* command can be used to force the new file into the editing buffer. If you use this command, however, all changes that have been made to the file currently in the buffer will be lost.

If more than one filename was listed when *ex* was invoked, you can use the *n* (*next*) command to move from filename to filename in the argument list. If you have not written the current file and changes have been made, an error message will be displayed.

As with the *s* and *w* commands (described later), the *n!* command variant forces the next file in the argument list into the buffer. Of course, when you do this, unsaved changes made to the current file are lost.

The *args* (*arguments*) command prints the files queued on the argument list, with the current argument delimited by the characters [and]. For example:

```
: args
file1 file2 [file3]
:
```

If you would like to go back and edit a file that you edited previously, use the *rew* (*rewind*) command to return to the beginning of the argument list. The "N files to edit" message will be displayed and the first file queued in the *args* list will be read into the editing buffer. The *rew!* command enables you to return to the previously edited file without saving changes made to the current file.

Writing the Buffer to a File

Use the *write* (*w*) command to write the editing buffer to a file. No changes are made to your file unless the buffer is written. Typing *w* writes the buffer to the file requested when you began the edit session. Typing *wq* saves your file and terminates the editing session. To write the contents of the editing buffer to a file other than the file you began editing, add a filename after the *w* on the command line. If the file exists, a message noting that fact will be displayed. If the *noterse* option is set, directions for coping with the situation will also be displayed.

When the *write* command is preceded by a single line number, only the line specified is written. Larger text segments are written by preceding the *write* command with the upper and lower boundaries of the segment. For example, the command:

```
: 10,60w foo
```

writes the lines from line 10 to line 60 to the file *foo*.

Two methods for overwriting existing files are available. Type either *w!* or *wa*.

To append text to the end of a file, use the characters >> as follows:

```
:w>>chaucer
```

Quitting the Editor

Use the *q* (*quit*) command to exit the editor. Several variants of this command exist; however, the *q* suffices for most situations. If you neglect to write your file before trying to exit, however, you will be warned by the editor:

```
:q
No write since last change (quit! overrides)
:
```

As the message indicates, you can use the command variant *q!* to exit *without* saving any changes into your file. The *q!* command also enables you to exit the editor without editing all the files in the *args* list.

Advanced Editing Operations

This section describes the use of editing commands which enable you to copy, transpose, mark, join, move, and delete text lines. In some cases, two different commands perform an operation identically. For the sake of brevity, only the more commonly used commands are discussed here.

Copying Lines of Text

There are two methods for copying text. The simpler method uses the *co* (*copy*) command to reproduce the lines of text after the specified address. For example:

```
: 3,4 co 0
And bathed every vein in such liquor
As has virtue to engender the flower;
When that April with his showers sweet
Has pierced the drought of March to the root
And bathed every vein in such liquor
As has virtue to engender the flower;
When, also, Zephyrus with his sweet breath
Has inspired in every wood and heath
```

As you will notice, this command sequence copied lines 3 and 4 at the beginning of the file. Note that the duplicated lines will be written to the first line *after* the address specified.

The second method of copying text, which uses the *yank* (*ya*) and *put* (*pu*) commands, is more complicated to use, but enables you both to copy text from one location to the other in the editing buffer, and to copy text from one file to another.

Use *yank* to copy the specified lines into a buffer. The buffer the lines are copied into is separate from the buffer being used to edit the current file, and the lines copied are not deleted from the original file.

Use *put* to copy the lines from the separate buffer into either the original file or any other file. For example, to copy the last line of *chaucer* to the beginning of a file, first *yank* the line, then issue a *put* command with no buffer specified. When you do not specify a buffer, the last deleted or yanked text is restored. The series of operations needed to accomplish this transfer is as follows:

```
: 6 ya
: pu
:
```

Copying text between files is complicated by the fact that any command issued between the *yank* and the *put* commands will affect the buffer. There are 26 named buffers, though, which *ex* commands do not affect unless you make a specific request. These buffers are named "a" through "z". An example of the use of named buffers is included below.

To copy the first three lines from a file named *chaucer* to a file named *foo*, first copy the lines into one of the named buffers, then write the buffer to *foo*. For example:

```
: ex chaucer
"chaucer" 6 lines, 254 characters
: 1,3 ya a
: e! foo
"foo" 0 lines, 0 characters
: pu a
```

Using an uppercase character to refer to a named buffer instructs the editor to append the new text to anything that may already be in the buffer. If you use the lowercase character, the new text will be written over the original contents of the buffer.

Moving Lines of Text

In *ex,* deleted lines do not evaporate; they are written into either a temporary or a named buffer. In either case, they are easily retrieved with the *pu* command. You can use the combination of the *delete* and *put* commands to move text between files or to move text within a file.

For example, to move a line from *chaucer* to *foo,* use the following operations:

```
:ex chaucer foo
2 files to edit
"chaucer" 6 lines, 254 characters
: 2
The drought of March has pierced to the root
:d m
And bathed every vein in such liquor
: w
"chaucer" 5 lines, 208 characters
: n
"foo" 0 lines, 0 characters
: pu m
The drought of March has pierced to the root
: wq
"foo" 1 lines, 46 characters
```

After initiating the editor, the next command deletes the second line of *chaucer.* The deleted line is placed in the named buffer "m". *Foo* is then copied into the editing buffer and the deleted line transfers to *foo* with the *put* command. The *q* command is then used to write *foo* and to exit the editor.

To move the first line of *chaucer* to the end of the file, use the following command sequence:

```
:ex chaucer
"chaucer" 6 lines, 254 characters
: 1
When that April with his showers sweet
: d a
The drought of March has pierced to the root
: 5
Has inspired in every wood and heath
: pu a
When that April with his showers sweet
: wq
"chaucer" 6 lines, 254 characters
```

The first line of *chaucer* is deleted and placed in the named buffer "a". After the current line is moved to line 5 (the new end of the file), the *put* command moves the deleted line to (the new) line 6. Use the *wq* command to write *chaucer* and to exit the editor.

Joining Lines of Text

The *j* command places text from a specified range of lines together on a single line. If you specify only one line address, then the line specified and the next line are joined. If you do not specify any addresses, the current line and the next line are joined.

The *J* variant automatically adjusts the white space between lines. This command adjusts white space at each junction to provide at least one blank character. *J* adds two blanks to sentences ending with a period. If one of the lines to be joined starts with the (character, then *J* leaves no spaces.

The *j!* variant performs the join operation with no white space processing; the joined lines are simply concatenated.

Searching for Character Strings

Use the / and *?* commands to locate character strings. The / command searches forward for the next occurrence of a character string which matches specified pattern text. For example, the command:

:/vein

locates and displays the next line which contains "vein". Similarly, the command:

:?April

searches backward from the current line until locating an occurrence of the string "April".

Using the *global* (*g*) command with / or *?* will locate and display all occurrences of a particular string. Any commands appended to *g* are executed when a matching string is located. For example, the command sequence:

:g /the/d

deletes all the lines which contain the string "the".

The variant *g!* searches for lines which *do not* match the pattern text. The following example finds all the lines in the file which do not contain the word "April."

:g!/April
Has pierced the drought of March to the root
And bathed every vein in such liquor
As has virtue to engender the flower;
When, also, Zephyrus with his sweet breath
Has inspired in every wood and heath

Files are searched for strings which begin or end with particular patterns when the meta-characters \< and \> are added to the command sequence. The \< command instructs the editor to look for strings which begin with a particular pattern. The \> character finds strings which end with particular patterns. For example, the command:

: g\<w

finds three lines containing words which begin with "w":

When that April with his showers sweet
When, also, Zephyrus with his sweet breath
Has inspired in every wood and heath

(In *ex*, a word is defined as any combination of letters, numbers, or underscore characters surrounded by characters which are not letters, numbers, or underscore characters.)

Modifying Global Searches

You can use two options--*ignorecase* and *wrapscan*--to modify the parameters used by the editor for global searches.

The *set ignorecase* (*ic*) option instructs the editor to consider uppercase and lowercase letters identical for string-searching operations. *Ignorecase* must be turned on with the *set* command since the default is *noic*.

Use *set wrapscan* to control how far *ex* will search for a specific string. Normally, *ex* searches for specific strings from the current line to the end of the editing buffer and then wraps around to search from the beginning of the buffer to the current line. If *wrapscan* is unset, the range of any given search is limited to the end of the buffer. The search will no longer wrap around the buffer, and a message will be displayed if the string is not located before reaching the end of the buffer.

Invoking UNIX Commands

A shorthand notation exists within *ex* which makes it possible to run UNIX commands from within the editor.

The *!* command sends the remainder of the line to the shell. This command enables you to format *chaucer* (for example) very easily:

```
: ! nroff chaucer
```

The symbol % denotes the current file. So, if you are editing *chaucer*, you can format the file by typing:

```
: ! nroff %
```

Use *!!* to repeat the last system command. So, for example, if you want to re-format *chaucer* after correcting your formatting errors, you can type:

```
: !!
```

If you are running *ex* with the standard option defaults, *ex* will warn you if you try to use the *!* command without having first written the buffer. Although this function will keep you from trying to process unchanged files, you may want to turn it off. Turn it off by typing:

```
: set nowarn
```

Use the *sh* (shell) command to suspend the editor temporarily if you want to run more than one UNIX command. After typing this command, the standard UNIX prompt appears. You can now execute any UNIX commands you wish. When you have finished, type EOF (normally ↑D) and you will return to the suspended edit session.

Switching from Ex to Vi

The only command presented thus far that allows intraline editing is *s*. If you need to make changes *within* a line of text, use the *vi* command to switch the editor into visual mode. In visual mode, *ex* functions exactly as *vi* does. The *vi* command takes location arguments in the following formats:

10vi Displays a screenful of text beginning at line 10

10vi Displays a screenful of text with line 10 at the bottom

10vi Displays a screenful of text with line 10 in the center

To exit from visual mode, type *:Q*.

Recovering from System Crashes or Terminal Disconnections

In most cases, you will receive mail when signing on after a system crash or terminal disconnection informing you that your files have been saved. To recover *chaucer*, for example, move to the directory you were in when the system crashed, and type the command:

```
ex -r chaucer
"chaucer" [Dated: Fri Dec 9 14:31:19] 6 lines, 254 characters
```

The saved files will contain *most* of the changes you had made before the system crashed. In all likelihood, some changes, especially text yanked or deleted to unwritten buffers, will be lost.

If you do not receive mail when signing on, try the procedure outlined above anyway. If you achieve no results, contact your site system manager.

4

Using Sed

Introduction

Unlike the editors previously discussed, *sed* is noninteractive. Using *sed* is analogous to batch processing: you can make no modifications or enhancements to the command file once you have invoked the editor.

Sed differs from *ex* and *vi* in its manner of text processing. Typically, text editing is accomplished by reading a file into a buffer and using editing commands to alter bits of the file iteratively. *Sed* uses neither an editing buffer nor individually applied editing commands. Instead, the file is passed through a "filter" of editing commands. Each line of text is compared to the set of *sed* commands. If any of the commands apply to the line, *sed* makes the change; if not, the line is not modified.

Sed is typically used to make global or complicated changes to large documents or to edit using shell scripts. It is seldom used for other purposes.

The general characteristics of *sed* include the following:

- *Sed* imposes no limits on file size since it uses no temporary files and since only a few lines of input reside in the core at any given time. In fact, the size of files which may be edited with *sed* is limited only by the maximum file size in CONVEX UNIX.
- *Sed* accepts command scripts that you prepare. This capability enables you to make editing changes more efficiently and to reuse editing scripts.
- *Sed's* major disadvantages are the lack of a relative-addressing capability and the lack of immediate verification that the edit has achieved the intended result.
- *Sed* copies standard input or files to standard output, performing one or more editing commands on each line before writing the file to output.
- *Sed* uses the same regular expression operators supported by *vi, ex,* and *ed* (see *ed*(1)).

Getting Started

The format for invoking *sed* is:

sed [**-n**] [**-e** *script*] [**-f** *sfile*] [*file*] ...

The *-e* option shown here causes the specified files to be edited according to a "script." A script consists of one or more lines which contain one editing command each. You usually create scripts for a specific application, although occasionally scripts can be borrowed from other users.

Scripts which are lengthy or intended for reuse are commonly stored in script files. Access these files via the *-f* option, which instructs *sed* to look for a script in *sfile*.

Note that if a command sequence contains only one *-e* option and no *-f* options, you can omit the *-e* flag. The *-n* option is used to suppress default output.

The *sed* editing commands must adhere to the form shown below:

[address [, address]] command [arguments]

If no addresses are given, *sed* applies the specified commands to each input line copied into the pattern space. If you specify one address, *sed* applies the specified command to that line. If you specify two addresses, *sed* applies the specified commands to the text lines inclusively bounded by the two addresses. Addresses can be either line numbers or patterns.

Arguments consist of one or more lines. Each argument line except the last must end with the backslash \ . Arguments which contain blanks should be enclosed by double quotation marks " ". The consistent use of quotation marks as safety devices is strongly encouraged.

Command-Line Flags

As the previous section illustrates, three flags are recognized on the command line:

-f used to direct *sed* to a file containing a list of editing commands

-e used when including editing commands on the input line

-n used for suppressing normal output

Each of these options is discussed in the following subsections.

Using the -f Option

The *-f* option is used to direct *sed* to a file containing an editing script. For example, to apply the script located in a file named *alter* to a file named *chaucer*, enter the following command sequence:

sed -f alter chaucer

The name of the file containing the editing commands *must* immediately follow the *-f* flag.

Sed by default copies the edited file to "standard output"--usually the terminal. In the example above, the edited version of *chaucer* would be displayed on the screen. Typically, however, output is more useful when it is directed to a file. To write *sed* output to a file, output is redirected with the > symbol, as in all shell commands. For example:

sed -f alter chaucer > newchaucer

Using the -e Option

There is no need to build a script file if only a few commands are going to be applied to the file. Instead, use the *-e* option to insert the commands. For example, to delete lines 30 through 50 of *chaucer*, enter the following command sequence:

sed -e "30,50d" chaucer > newchaucer

If more than one editing script is included on the command line, precede each script with the *-e* flag:

sed -e "30,50d" -e "s/When/Wham/g" chaucer > newchaucer

In this example, *sed* deletes lines 30 through 50 of *chaucer*, and substitutes the string "Wham" for the string "When" each time "When" appears in the file.

The *-f* and *-e* flags can be used on the same command line.

Using the -n Option

In normal operation, *sed* modifies input lines according to the edit operations requested, then copies each modified line to standard output. The *-n* option is used to suppress this normal output mode. When you use *-n*, only those lines requested with the *p* (*print*) argument are displayed. For example, the following command sequence:

sed -n -e "s/When/Wham/" -e "/Wham/p" chaucer

displays only lines containing the string "Wham". In this case,

Wham that April with his showers sweet
Wham, also, Zephyrus with his sweet breath

is displayed. (Remember to use *-e* flag each time an editing command is invoked.) Notice that the line is displayed *after* it has been modified in accordance with all the editing commands. *Sed* works with files strictly in the order it is commanded to. The command sequence:

sed -n -e "/Wham/p" -e "s/When/Wham/" chaucer

would not result in any output since at the time the *p* (print) command was invoked, the string "Wham" would not exist in the file.

The *-n* option can also be used to display parts of files. For example, the command sequence:

sed -n "1,3p" chaucer > newchaucer

causes the first three lines of *chaucer* to be written to *newchaucer*. (Since only one editing command is used, the *-e* flag is unnecessary.)

Basic Functions

This section presents examples of commonly used editing functions. Refer to the *The CONVEX UNIX Programmer's Manual* for instructions on using other *sed* commands.

Adding More Text to an Existing File

Use the *a* command to append text to existing files. To add text to the file *chaucer*, for example, use the following procedure:

1. Create a script file which contains both the *a* command and the text to be appended. For example, you might use the command sequence:

```
$a\
ho ho ho ho ho ho \
ho ho ho ho ho ho \
ho ho ho ho ho ho
```

2. Invoke *sed* with the *-f* option and the name of the script file, as follows:

```
sed -f hoho chaucer > newchaucer
```

Note that the backslash character \ is a continuation character added to the end of each text line (except the last one). The $ symbol means "last line" and is used to add text to the end of the file.

Substituting Simple Patterns

Several examples of text substitution have been shown previously. As you may recall, the *s* command is used to substitute patterns. For instance, the command sequence

```
sed -e "1,6s/April/May/" chaucer
```

substitutes "May" for "April" the first time the string appears in any of lines 1 through 6. Normally, the *s* command would modify only the first occurrence of the string "April" on each line. To modify each occurrence, append the *g* (global) flag at the end of the *s* command.

Deleting Text

Text is deleted with the *d* command. The command sequence:

```
sed "1,20d" chaucer > newchaucer
```

causes the first 20 lines of *chaucer* to be deleted, and writes the file to *newchaucer*.

Printing Text

Printing is accomplished by means of the *p* command. The command sequence:

```
sed "1,10p" chaucer
```

causes the first 10 lines of the file *chaucer* to be displayed on the terminal.

Replacing Text Throughout a File

The *y* (*transform*) command is used to replace text. The standard format of the command is:

```
sed -e "y/string1/string2/"
```

The command causes all occurrences of the characters in *string1* to be replaced with the corresponding characters in *string2*. The lengths of *string1* and *string2* must be equal.

For example, to replace in the file *foo* every occurrence of the string "mumble" with "jumble," use the following command sequence:

```
sed "y/mumble/jumble/" foo
```

Using Regular Expressions

Regular expressions are special patterns that text editors (and other programs) use to search for text strings. The use of regular expressions enables you to:

- Construct a search pattern that matches more than one string
- Construct a simply typed search pattern that matches a complex string

The basic use of regular expressions is illustrated in the following examples. To convert all instances of the strings "Convex" and "convex" to "Convex-1" in the file 01.s, use the regular expression /[Cc]onvex/ as follows:

```
sed -e "s/[Cc]onvex/Convex-1/g" 01.s
```

Double quotes enclose the regular expression to protect the brackets from interpretation by the shell's file-matching mechanism.

The command sequence shown below is used to locate and expand all occurrences of the string "Convex Computer Corp".

```
sed -e "s/Con.*rp/Convex Computer Corporation/g"
```

In the command sequence above, the characters .* match an indeterminate number of characters after the string "Con", thus eliminating the need to type the longer sequence "Convex Computer Corp". Note that the regular expression "Con.*rp" also matches the strings "Confused carp" and "Conscientious objectors play harp"--strings unlikely to appear in most documents.

Index

W