Editing Text Files

Contents

Contents — *Continued*

Tables

Preface

This manual describes the facilities for editing and manipulating text available with standard Sun system software. The preliminary chapters cover the use of various editors, and later chapters introduce more advanced text manipulation commands. We assume you are familiar with a terminal keyboard and the Sun system. If you are not, read the *SunOS User's Guide: Getting Started* .

Summary of Contents

This manual is divided into eight chapters. The first few chapters cover the standard text editors, presented in decreasing order of use. Remaining chapters describe how to scan and manipulate files. Here is a short outline of the chapters:

□ The introductory chapter provides a guide to the available editing tools, and suggestions for what to if something goes wrong in the editor. Newcomers to editing on a Sun workstation should start here.

□ The chapter on `vi` provides tutorial and reference information on the most commonly used visual display editor.

□ The next chapter constitutes a command reference for the line-oriented editor `ex`, which is actually the same editor (in a different mode) as `edit` and `vi`.

□ The chapter on `ed` provides a user's guide to the oldest UNIX system editor, which is now largely outmoded.

□ The chapter on scanning files is divided into two parts: looking at files (as with `more`) and searching through files (as with `grep`).

□ The chapter on `awk` presents an interesting programming language designed for pattern scanning and text processing.

□ The chapter on manipulating files is divided into three parts: comparing files (as with `diff`), modifying files (as with `sort`), and printing files (with `lpr`).

Usage Conventions

Throughout this manual we use

```
hostname%
```

as the prompt after which you type system commands. **Bold typewriter font** indicates commands that you must type exactly as printed in the manual.

Regular typewriter font represents what the system prints out to your screen. Typewriter font also specifies Sun system command names (program names) and illustrates source code listings. *Italic font* indicates general arguments or parameters that you should replace with a specific word or string. We also occasionally use italics to emphasize important terms.

Reading Suggestions

The information in this manual is presented in two styles: examples then exercises, and reference. As you are reading sections of this manual, sit down at your workstation and try the exercises and examples. The reference sections serve two functions: they explain in greater detail how to use the most-often-used features, and also cover the less-often-used features and options. Another reference source for additional details on Sun system commands and programs is the *SunOS Reference Manual.*

Introduction to Text Editing

If you are familiar with text editors on the UNIX system, you can refer directly to the chapter in this manual covering the specific editor in which you are interested.

If you want a quick reference to remind you of a particular feature of either vi or ex, you can refer to one of the summary sheets for these editors, following the vi and ex chapters, respectively.

If you are *not* familiar with available text editors, read the chapter "Editing Files" in *SunOS User's Guide: Getting Started* . It provides a good introduction to vi, including how to create a file, move the cursor around within a file, and change the contents of a file.

1.1. Available Editors

SunOS provides the text editors vi, ex, edit, ed, and textedit. The one you will probably use most often is vi, although you may prefer another editor that is more familiar to you. When inside the window system, textedit provides a mouse-based editing interface; for an introduction to this editor see the *SunView User's Guide*.

vi is a screen-oriented, or display editor. It is "built on top of" ex, which means vi has almost all of the features of ex, plus many more features that are specific to vi. You can issue almost all ex commands from within vi by preceding them with the colon (:) character. Only a few ex commands require you to invoke the ex editor explicitly. edit is a subset of ex, and is therefore covered in that chapter.

The following diagram briefly sketches the history of UNIX system text editors. ed was the original line editor, after which all the others were modeled. ex improved on ed, by being less terse, and providing display options like numbered lines, by allowing shorthand versions of commands, and by responding with clearer error messages. However, ex is still a line editor. edit, another line editor, provides a subset of ex's features. Later, *open mode* was added to ex, which enabled the user to make changes and move the cursor around on only a single line at a time. The most advanced and most widely-used text editor as of this writing is the screen-oriented editor vi.

```
┌────┐
│ ed │     terse, hard to use
└────┘
     \
      \
       ↘
      ┌────┐     less terse than ed
      │ ex │     Options
      └────┘     Shorthand
       /  \      Better Error Messages
      /    \
     ↙      \
┌──────────────┐ \
│     edit     │  ↘
│ (subset of ex)│  ┌──────────────────┐   Changes and move around
└──────────────┘  │ open mode of ex  │   on single line
                  └──────────────────┘
                           \
                            \
                             ↘
                          ┌────┐
                          │ vi │   visual editor
                          └────┘
```
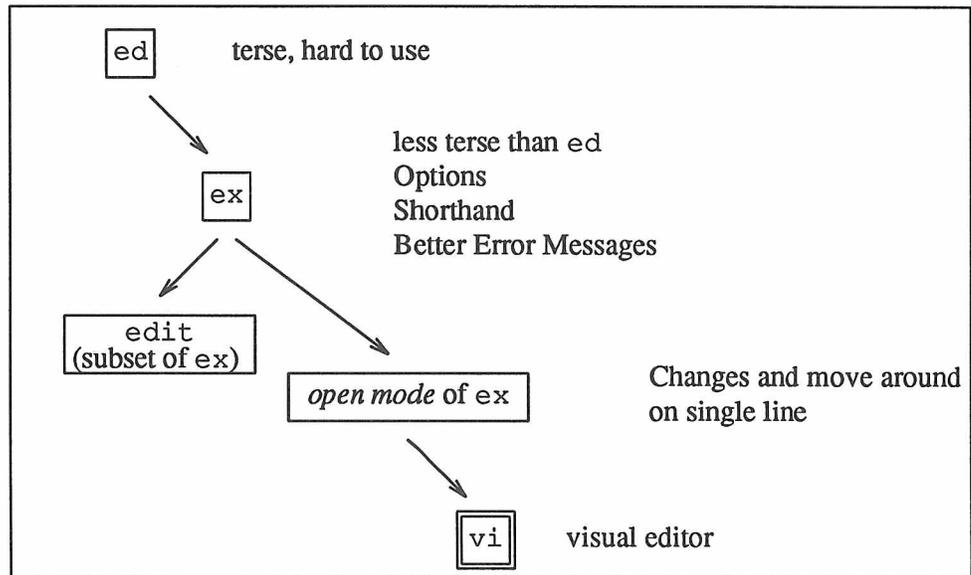
1.2. What to Do If Something Goes Wrong

In case you make a mistake, or something goes wrong, here are some suggestions for what to do.

If you make a mistake in the editor that you cannot recover from easily, don't panic. As long as you don't *write* the file *and quit* the editor, you can retrieve the original file. In such a case you have two choices:

1.   Write ALL (good and bad) changes you just made to a new file and quit editing the current file, or

2.   Quit the editor without saving any changes you just made.

You should write the entire edit buffer into a new file as long as most of the new changes are valid. In vi, the commands are :w *newfilename* to write the new file, then :q! to get out of the editor. You should quit without saving ANY of the changes if you know they are all wrong. In vi, the command is :q!.

Occasionally, you can get into a state where your workstation or terminal acts strangely. For example, you may not be able to move the cursor, or your cursor may disappear, or the terminal won't echo what you type, or typing RETURN may not cause a linefeed or return the cursor to the left margin. After a suitable length of time, try the following solutions:

□   First, type CTRL-Q. In case you had accidentally typed CTRL-S, freezing the screen, this would resume the suspended output. Typing CTRL-Q would also resume suspended output from accidentally typing a NO SCRL key on your keyboard (also called SET UP/NO SCROLL on some terminals). This also freezes the keyboard like typing a CTRL-S.

□   Next, try pressing the LINEFEED key, followed by typing RESET, and pressing LINEFEED again.

□   If that doesn't help, try logging out and logging back in. If you are using a terminal, try powering it off and on to regain normal operation.

□   If you get unwanted messages or garbage on your screen, type CTRL-L to refresh the workstation screen. (Use CTRL-R on a terminal.)

If your system goes down, the edit buffer is automatically saved in a file. Depending on the elapsed time since your last change, most to all of your latest changes are recoverable. After rebooting your system, or doing whatever needs to be done, you will receive mail indicating that the file has been saved. The mail message contains instructions on how to recover the file with your edits in it. First, return to the directory where the file belongs. Then, re-enter the editor with the −r option to *restore* the file:

```
hostname% vi −r filename
```

When you are ready, write the changes to the file by typing `:w` or `:wq`.

1.3. Other Text-Handling Programs

Other utility programs such as `awk`, `sed`, `grep`, `fgrep`, `egrep`, and `tr` operate on a text file, but do not change the original file. You pass the file to be "edited" through a *script* (such as `awk` or `sed`) or *command* (such as `grep`) and the "changes" appear on your screen, but the file remains intact. Refer to the following diagram for an outline of how these utility programs work.
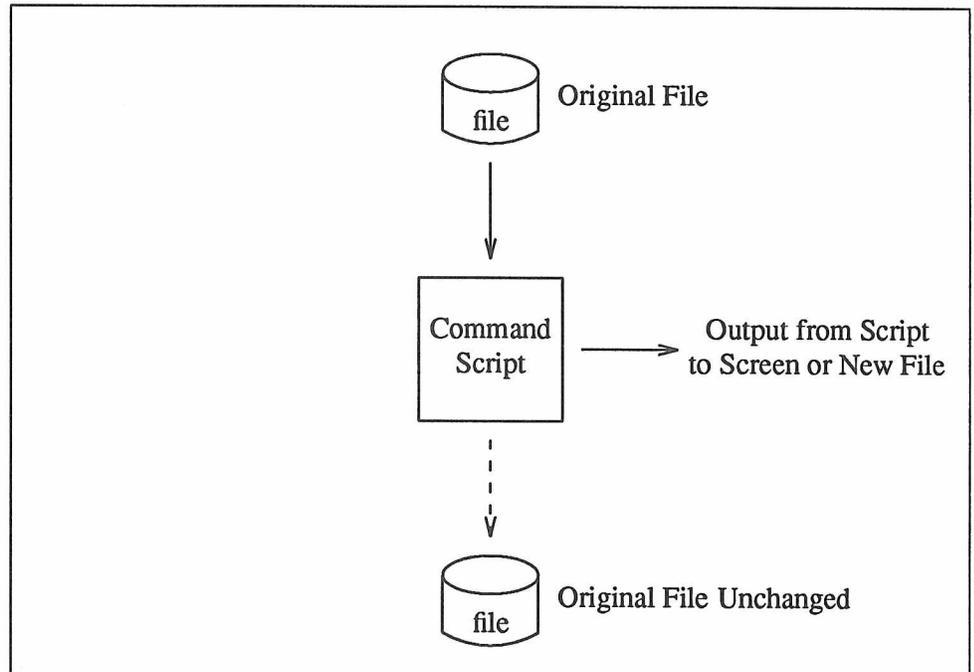
For more information, refer later chapters in this manual.

Using vi, the Visual Display Editor

This chapter describes vi (pronounced *vee-eye*), the visual display editor.[1] The first part of this chapter discusses the basics of using vi. The second part gives a command reference and provides information on setting up terminals. Finally, there is a quick reference sheet, which summarizes vi commands. Keep this reference sheet handy while learning vi.

This chapter assumes you are using vi on the Sun Workstation. If you are using vi on a terminal, refer to the section "Terminal Information" for instructions on setting up your terminal.

2.1. vi and ex

Actually vi and ex are links to the same program. Depending on whether you invoke the vi or ex command, you see different aspects of this program. While ex presents a line-oriented editing interface, vi presents a screen-oriented interface. Of course, the full command set of ex is available from within vi, and screen editing is available from within ex. Most people prefer editing with vi rather than with ex. However, some editing tasks (such as global substitutions) are performed more easily with ex. See the chapter "Command Reference for the ex Line Editor" for more informatio. n the ex command set.

After typing vi, you are placed in visual n.   after typing ex, you are placed in command mode. It is possible to switch b ween these modes without leaving the editor. When you are in line-oriented command mode, you can escape to visual mode by typing **vi** (Return). When you are in visual mode, you can escape to line-oriented command mode by typing **Q** all by itself. Alternatively, you can type **:** all by itself, which places the cursor on the command line at the bottom of the screen. From here you can issue any single-line ex command, ending it with a (Return). All the colon-commands introduced in the section entitled "File Manipulation Commands" are also available in ex.

In rare instances, an internal error may occur in vi. In such a case you will get a diagnostic and be left in the command mode of ex. You should save your work and quit by giving the command **wq** (Return) after the **:** prompt that the editor gives you. Or if you prefer, you can re-enter visual mode by giving the command **vi** (Return) after the colon prompt.

---

[1] The material in this chapter is derived from *An Introduction to Display Editing with Vi*, W.N. Joy, M. Horton, University of California, Berkeley and *Vi Command and Function Reference*, A.P.W. Hewett, M. Horton.

2.2. Getting Started

When using vi, changes you make to the file you are editing are reflected in what you see on your workstation screen.

During an editing session, there are two usual modes of operation: *command* mode and *insert* mode. In command mode you can 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, like d for delete and c for change, are combined with the motion commands to form operations such as delete word or change paragraph. You can do other operations that do not involve entering fresh text. To enter new text into the file, you must be in insert mode. You get into insert mode with the a or A (append), o or O (open) and I or i (insert) commands. You get out of insert mode by typing the ESC (escape) key (or ALT on some keyboards). The significant characteristic of insert mode is that commands can't be used, so anything you type except ESC is inserted into the file. If you change your mind anytime in insert mode using vi, typing ESC cancels the command you started and reverses to command mode. If you had already typed some characters, typing u *undoes* the last insert operation. Also, if you are unsure of which mode you are in, type ESC until the screen flashes; this means that you are back in command mode.

Run vi on a copy of a file you are familiar with while you are reading this. Try the commands as they are described.

Editing a File

To use vi on the file, type:

```
hostname% vi filename
```

replacing *filename* with the name of the file copy you just created. The screen clears and the text of your file appears.

If you do not get the display of text, you may have typed the wrong filename. vi has created a new file for you with the indication "file" [New file]. Type :q (colon and the 'q' key) and then type the RETURN key. This should get you back to the command level interpreter. Then try again, this time spelling the filename correctly.

If vi doesn't seem to respond to the commands you type here, try sending vi an interrupt by typing a CTRL-C (or INTERRUPT signal) at your workstation (or by pressing the DEL or RUB keys on your terminal). Then type the :q command again followed by a RETURN. If you are using a terminal and something else happens, you may have given the system an incorrect terminal type code. vi may make a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. Type a :q and RETURN. Figure out what you did wrong (ask someone else if necessary) and try again.

The Editor's Copy — Editing in the Buffer

vi does not directly modify the file you are editing. Rather, vi makes a copy of this file in a place called the *buffer*, and remembers the file's name. All changes you make while editing only change the contents of the buffer. You do not affect the contents of the file unless and until you *write* the buffer back into the original file.

Arrow Keys

The editor command set is independent of the workstation or terminal you are using. On most terminals with cursor positioning keys, these keys will also work within the editor.[2] If you don't have cursor positioning keys, that is, keys with arrows on them, or even if you do, you can use the h, j, k, and l keys as cursor positioning keys. As you will see later, h moves back to the left (like CTRL-H, a backspace), j moves down (in the same column), k moves up (in the same column), and l moves the cursor to the right.

Special Characters: ESC, CR and CTRL-C

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 on some terminals). It is near the upper left corner of your workstation keyboard. Try typing this key a few times. vi flashes the screen (or beeps) to indicate that it is in a quiescent state. You can cancel partially formed commands with ESC. When you insert text in the file, you end the text insertion with ESC. This key is a fairly harmless one to press, so you can just press it until the screen flashes if you don't know what is going on.

Use RETURN (or CR for *carriage return*) key to terminate certain commands. It is at the right side of the workstation keyboard, and is the same key used at the end of each shell command.

Use the special character CTRL-C (or DEL or RUB key), to send an interrupt, to tell vi to stop what it is doing. It is a forceful way of making vi listen to you, or to return vi to the quiescent state if you don't know or don't like what is going on.

Try typing the '/' key on your keyboard. Use this key to search for a string of characters. vi displays the cursor at the bottom line of the screen after a '/' is displayed as a prompt. You can get the cursor back to the current position by pressing BACK SPACE (or DEL); try this now. This cancels the search. Typing CTRL-C also cancels the search. From now on we will simply refer to typing CTRL-C (or pressing the DEL or RUB key) as 'sending an interrupt.'[3]

vi often echoes your commands on the last line of the screen. If the cursor is on the first position of this last line, then vi is performing a computation, such as locating 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 vi by sending an interrupt.

---

[2] Note for the HP2621: on this terminal the function keys must be *shifted* to send to the machine, otherwise they only act locally. Unshifted use leaves the cursor positioned incorrectly.

[3] 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.

Getting Out of vi — :q :q!
:w ZZ :x :wq

When you want to get out of vi and end the editing session, type :q to *quit*. If you have changed the buffer contents and type :q, vi responds with No write since last change (:quit! overrides). If you then want to quit vi without saving the changes, type :q!. You need to know about :q! in case you change the editor's copy of a file you wish only to look at. Be very careful not to give this command when you really want to save the changes you have made.

Do not type :q! if you *want* to save your changes. To save or *write* your changes without quitting vi, type :w. While in the middle of an editing session, if you are sure about the changes you have made, it's a good idea to save your changes from time to time by typing :w.

To write the contents of the buffer back into the file you are editing, saving any changes you have made, and then to quit, type ZZ or :x. And finally, to write the file even if no changes have been made, and exit vi, type :wq.

You can terminate all commands that read from the last display line with an ESC as well as a RETURN.

2.3. Moving Around in the File

vi has a number of commands for moving around in the file. You can *scroll* forward and backward through a file, moving part of the text on the screen. You can *page* forward and backward through a file, by moving a whole screenful of text. You can also display one more line at the top or bottom of the screen.

Scrolling and Paging —
CTRL-D CTRL-U CTRL-E
CTRL-Y CTRL-F CTRL-B

The most useful way to move through a file is to type the control (CTRL) and D keys at the same time, sending a CTRL-D. We use this notation to refer to control sequences from now on. When coupled with the CTRL key, the shift key is ignored, so CTRL-D and CTRL-d are equivalent.

Try typing CTRL-D to see that this command scrolls *down* in the file. The command to scroll *up* is CTRL-U. (Many dumb terminals cannot scroll up at all. In that case type CTRL-U to clear and refresh the screen, placing a line that is farther back in the file at the top of the screen.)

If you want to see more of the file below where you are, you can type CTRL-E to expose one more line at the bottom of the screen, leaving the cursor where it is. The CTRL-Y (which is hopelessly non-mnemonic, but next to CTRL-U on the keyboard) exposes one more line at the top of the screen.

You can also use the keys CTRL-F and CTRL-B to 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 CTRL-D and CTRL-U if you wish. CTRL-F and CTRL-B also take preceding counts, which specify the number of pages to move. For example, 2CTRL-F pages forward two pages.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, typing CTRL-F to page forward leaves you only a little context to look back at. Scrolling with CTRL-D on the other hand, leaves more context, and moves more smoothly. You can continue to read the text as scrolling is taking place.

Searching, Goto, and Previous Context — / ? G

Another way to position yourself in the file is to give vi a string to search for. Type the character '/' followed by a string of characters terminated by RETURN. vi positions the cursor at the next occurrence of this string. Try typing n to go to the *next* occurrence of this string. The character '?' searches backward from where you are, and is otherwise like '/'. N is like n, but reverses the direction of the search.

You can string several search expressions together, separated by a semicolon in visual mode, the same as in command mode in ex. For example:

```
/today/;/tomorrow
```

moves the cursor to the first 'tomorrow' after the next 'today'. This also works within one line.

These searches normally wrap around the end of the file, so you can find the string even if it is not on a line in the direction you search, provided it is somewhere else in the file. You can disable this *wraparound* with the command :se nowrapscanCR, or more briefly :se nowsCR.

If the search string you give vi is not present in the file, vi displays Pattern not found on the last line of the screen, and the cursor is returned to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with a caret character (^). To match only at the end of a line, end the search string with a dollar sign ($). So to search for the word 'search' at the beginning of a line, type:

```
/^search<CR>
```

and to search for the word 'last' at the end of a line, type:

```
/last$<CR>
```

Actually, the string you give to search for here can be a *regular expression* in the sense of the editors ex and ed. If you don't wish to learn about this yet, you can disable this more general facility by typing

```
:se nomagic<CR>
```

By putting this command in EXINIT in your environment, you can have always this *nomagic* option in effect. See the section on "Special Topics" for details on how to do this.

The command G, when preceded by a number, positions the cursor at that line in the file. Thus 1G moves the cursor to the first line of the file. If you do not give G any count, it positions you at the last line of the file.

If you are near the end of the file, and the last line is not at the bottom of the screen, vi places only the character tilde (~) 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 CTRL-G. vi shows 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 characters already displayed from the buffer. For example:

```
"data.file" [Modified] line 329 of 1276 —8%—
```

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 with the line number to get back where you were.

You can get back to a previous position by using the command ' ' (two apostrophes). This returns you to the first non-blank space in the previous location. You can also use ` ` (two back quotes) to return to the previous position. The former is more easily typed on the keyboard. This is often more convenient than G because it requires no advance preparation. Try typing a G or a search with / or ? and then a ` ` to get back to where you were. If you accidentally type n or any command that moves you far away from a context of interest, you can quickly get back by typing ` `.

Moving Around on the Screen — h j k l + - H M L

Now try just moving the cursor around on the screen. Try the arrow keys as well as h, j, k, and l. You will probably prefer these keys to arrow keys, because they are right underneath your fingers. 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 scrolls down (and up if possible) to bring a line at a time into view.

Type the + key. Each time you do, notice that the cursor advances to the next line in the file, at the first non-blank position on the line. The − key is like + but the cursor goes to the first non-blank character in the line above.

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 takes you to the top (*home*) line on the screen. Try preceding it with a number as in 3H. This takes you to the third line on the screen. 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, so 5L takes you to the fifth line from the bottom.

Moving Within a Line — b w e E B W

Now pick a word on some line on the screen, not the first word on the line. Move the cursor using h, j, k, l or RETURN and − to be on the line where the word is. Try typing the w key. This advances the cursor to the next *word* on the line. W advances to the next word ignoring any punctuation. Try typing 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 BACKSPACE (or CTRL-

H) key which moves left one character. The key h works as CTRL-H does and is useful if you don't have a BACKSPACE key.

If the line had punctuation in it, you may have noticed that the w and b keys stopped at each group of punctuation. You can also go backward and forward words without stopping at punctuation by using W and B rather than the lower case equivalents. You can think of these as bigger words. The E command advances to the end of the current word, but unlike e, ignores punctuation. Try these on a few lines with punctuation to see how they differ from the lower case e, 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 typing w.

Viewing a File — view

If you want to use the editor to look at a file, rather than to make changes, use view instead of vi. This sets the *readonly* option which prevents you from accidently overwriting the file. For example, to look at a file called *kubla*, type:

```
hostname% view kubla
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.
"kubla" [Read only] 5 lines, 149 characters
```

To scroll through a file longer than one screenful, use the characters described in the previous section on "Scrolling and Paging." To get out of view, type :q. If you accidentally made changes to the file while the *readonly* option was set, type :q! to exit.

2.4. Making Simple Changes

Simple changes involve inserting, deleting, repeating, and changing single characters, words, and lines of text. In vi, you can also undo the previous change with ease in case you change your mind.

Inserting — i I a A o and O

There are two basic commands for inserting new text: i to *insert* text to the left of the cursor, and a to *append* text to the right of the cursor. After you type i, everything you type until you press ESC is inserted into the file. Try this now; position yourself at 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 type ESC.)

Now try finding a word that can, but does not, end in an 's'. Position the cursor at this word and type e (move to end of word), then a (for append), 's', and ESC to terminate the text insert. Use this sequence of commands to easily make a word plural.

Try inserting and appending a few times to make sure you understand how this works.

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 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 or up, while the lower-case key moves forward 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, type 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, dumb terminal vi 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 that would occur if vi attempted to always keep the tail of the screen up to date. The tail of the screen will be fixed up, and the missing lines will reappear, when you type ESC.)

While you are inserting new text, you can use the DEL key at the system command level to backspace over the last character you typed. (This may be CTRL-H on a terminal.) Use CTRL-U (this may be CTRL-X on a terminal) to erase the input you have typed on the current line. In fact, the character CTRL-H (backspace) always works to erase the last input character here, regardless of what your erase character is.

CTRL-W erases a whole word and leaves you after the space after the previous word; use it to quickly back up when inserting.

Notice that when you backspace during an insertion, the characters you backspace over are not erased; the cursor moves backward, 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 you press 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 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.

Making Small Corrections —
x r s R

You can make small corrections in existing text quite easily. Find a single character that 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 with h (or the BACKSPACE key or CTRL-H) or type a SPACE (using the space bar) until the cursor is on the character that is wrong. If the character is not needed, type 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 a single character is incorrect, you can *replace* it with the correct character by typing the command r*c*, where *c* is replaced by the correct character. You don't need to type ESC. If you want to replace or type over more than one character, type R and then the ESC key to get out of insert mode when you are finished. Finally, if the incorrect character should be replaced by more than one character, type s which *substitutes* for the single character, a string of characters, and end the substitution with ESC. If there are a small number of incorrect characters, you can precede s with a count of the number of characters to be replaced. You can use counts with x to specify the number of characters to be deleted and with r, such as 4rx to specify that a character be replaced with four x's.

Use xp to correct simple typos in which you have inverted the order of two letters. The p for *put* is described later.

Deleting, Repeating, and Changing — dw . db c

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 typing ' . ' a few times. Notice that this repeats the effect of the dw. The ' . ' repeats the last command that made a change. You can remember it by analogy with an ellipsis ' . . . '.

Now try db. This deletes a word before the cursor, namely the preceding word. Try dSPACE. This deletes a single character, as does the x command.

Use D to delete the rest of the line the cursor is on.

Another very useful operator is c or *change*. Thus cw changes the text of a single word. You follow it by the replacement text ending with an ESC. Find a word that you can change to another, and try this now. Notice that the end of the text to be changed is marked with the dollar sign character ($) so that you can see this as you are typing in the new material.

Operating on Lines — dd cc S

It is often the case that you want to operate on lines. Find a line you want to delete, and type dd, the d operator twice. This deletes the line.

If you are on a dumb terminal, vi may just erase the line on the screen, replacing it with a line with only an at-sign (@) 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 changes a whole line, erasing its previous contents and replacing them with text you type up to an ESC. The command S is a convenient synonym for cc, by analogy with s. Think of S as a substitute on lines, while s is a substitute on characters.

You can delete or change more than one line by preceding the dd or cc with a count, such as 5dd, which 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.[4] Notice that vi lets you know when you change a large number of lines so

---

[4] One subtle point here involves using the ' / ' search after a d. This normally deletes characters from the current position to the point of the match. If what is desired is to delete whole lines including the two points,

sun
microsystems

that you can see the extent of the change. It also always tells you when a change you make affects text you cannot see.

Undoing — u U

Now suppose that the last change 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, vi provides a u command to *undo* the last change you made. Try this a few times, and give it twice in a row to notice that a 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, only as long as you do not move the cursor off the line. If you move the cursor away from the line you changed, U does nothing.

You can recover text that you delete, even if u (undo) will not bring it back; see the section on "Recovering Lost Lines" on how to recover lost text.

2.5. Rearranging and Duplicating Text

This describes more commands for moving in a file and explains how to rearrange and make copies of text.

Low-level Character Motions — f F ^

Now move the cursor to a line where there is a punctuation or a bracketing character such as a parenthesis, a comma or a period. Try the command fx to *find* the next x character to the right of the cursor in the current line. Try then hitting a ; which finds the next instance on that line 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 an F command, which is like f, but searches backward. After instituting a search, the ; repeats the search in the same direction as it was begun, and a comma (,) repeats the search in the opposite direction.

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,* that is, 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, a ' ^ ' moves the cursor to the first non-blank position on the line, and a $ moves it to the end of the line. Thus $a appends new text at the end of the current line (as does A which is easier to type).

Your file may have tab (CTRL-I) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every eight positions.[5] When the cursor is at a tab, it sits on the last of the several spaces that represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

---

give the pattern as /*pat*/+0, a line address.

[5] You can set this with a command of the form : se ts=*x*<CR>, where *x* is four to set tabstops every four columns, for example. This affects the screen representation within the editor.

On rare occasions, your file may have non-printing characters in it. These characters are displayed as control sequences, and look like a caret character (^) adjacent to another character. For example, the symbol for a new page (CTRL-L), looks like ^L in the input file. However, spacing or backspacing over the character reveals that the two characters displayed represent only 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 CTRL-V before the control character. The CTRL-V quotes the following character, causing it to be inserted directly into the file.

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) deletes the rest of the current sentence; likewise d (deletes 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 as ending at a '.', '!' or '?' 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 the line.

The operations ' {' and '} ' move over paragraphs and the operations ' [[' and ']] ' move over sections. The ' [[' and ']] ' operations require the operation character to be doubled because they can move the cursor far 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 type accidentally.

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 macro package, that is the .IP, .LP, .PP, and .QP macros. You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See the section on "Special Topics" for details. The .bp directive is also considered to start a paragraph. Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands take counts to operate over groups of sentences and paragraphs.

Sections in the editor begin after each macro in the *sections* option, normally .NH and .SH, and each line with a formfeed CTRL-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 view size in which to redraw the screen at the new location, and this size is the base size for newly-drawn screens 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 screen area.)

Rearranging and Duplicating Text — y Y p P

vi has a single unnamed buffer where the last deleted or changed text is saved away, and a set of named buffers a–z that 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 that 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 you yank forms a part of a line, or is an object such as a sentence that 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 an o or O command.

Try the command YP. This makes a copy of the current line and leaves the cursor 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 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 vi 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 must use a named buffer.

2.6. High-Level Commands

A description of high-level commands that do more than juggle text follows.

Writing, Quitting, and Editing New Files — ZZ :w :q :e :n

So far you have seen how to enter vi and to write out your file using either ZZ or :wCR. The first exits from vi, writing if changes were made, and the second writes and stays in vi. We have also described that 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, you type

```
:q!<CR>
```

to *quit* from the editor without writing the changes.

You can also re-edit the same file and start over by typing :e!CR. Use the '!' command 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 also edit a different file without leaving vi by giving the command
:e *name*CR. If you have not written out your file before you try to do this, vi
tells you this, ('No write since last change: (:edit! overrides)') and delays editing
the other file. You can then type :wCR to save your work, followed by 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
save changes automatically, include set autowrite in your EXINIT, and use
:n instead of :e. See the "Special Topics" section for details on EXINIT.

**Escaping to a Shell — :! :sh
CTRL-Z**

You can get to a shell to execute a single command by giving a vi command of
the form :!*cmd*CR. The system runs the single command *cmd* and when the
command finishes, vi asks you to Press RETURN to continue. When
you have finished looking at the output on the screen, type RETURN, and vi
redraws the screen. 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, give the command
:shCR. This gives you a new shell, and when you finish with the shell, ending it
by typing a CTRL-D, vi clears the screen and continues.

Use CTRL-Z to suspend vi and to return to the top level shell. The screen is
redrawn when vi is resumed. This is the same as :stop.

Marking and Returning — m

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 type ` a. The cursor
will return to the place you marked. Marks last only until you edit another file.

When using operators such as d and referring to marked lines, it is often desir-
able 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-blank character of the marked line;
similarly ' ' moves to the first non-blank character of the line containing the pre-
vious context mark ` `.

**Adjusting the Screen CTRL-L,
z**

If the screen image is messed up because of a transmission error to your worksta-
tion, or because some program other than vi wrote output to your workstation,
you can type a CTRL-L, the ASCII form-feed character, to refresh the screen. (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 CTRL-R to *retype* the screen,
closing up these holes.[6])

If you wish to place a certain line on the screen at the top middle or bottom of the
screen, position the cursor to that line, and give a z command. Follow the z
command with a RETURN if you want the line to appear at the top of the

---

[6] This includes Televideo 912/920 and ADM31 terminals.

window, a '.' if you want it at the center, or a '–' if you want it at the bottom.

If you want to change the window size, use the z command as in z 5<CR> to change the window to five lines.

2.7. Special Topics

There are several facilities that you can use to customize an editing session.

Options, the Set Variable, and Editor Start-up Files

vi has a set of options, some of which have been mentioned above. The most useful options are described in the following table.

Table 2-1   *Editor Options*

Option	Default	Description
autoindent	noai	Supply indentation automatically
autowrite	noaw	Automatic write before :n, :ta, CTRL-^, !
ignorecase	noic	Ignore letter case in searching
lisp	nolisp	({) } commands deal with S-expressions
list	nolist	Tabs print as ^I, end of lines marked with $
magic	magic	The characters . [and * are special in scans
number	nonu	Lines displayed prefixed with line numbers
paragraphs	para=IPLPPPQPP LI...	Macro names that start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one
sections	sect=NHSHH HU...	Macro names that start new sections
shiftwidth	sw=8	Shift distance for <, > and input CTRL-D
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
```

Toggle options can be set or unset by statements of one of these forms:

```
set opt
set noopt
```

Put these statements in your environment variable EXINIT (described below), or use them while you are running vi by preceding them with a : and following them with a RETURN. For example, to display line numbers at the beginning of each line, use:

```
:se nu
```

To get a list of all options that you have changed, or the value of a single option, use:

```
:set<CR>
redraw term=sun wrapmargin=8

:set opt?<CR>
```

For example:

```
:set noai?<CR>
noautoindent
```

This command generates a list of all possible options and their values:

```
:set all<CR>
```

You can abbreviate set to se. You can also put multiple options on one line, as follows:

```
:se ai aw nu<CR>
```

When you set options with the set command, they only last until you terminate the editing session in vi. It is common to want to have certain options set whenever you use the editor. To do this, create a list of ex commands to be run every time you start up vi, ex, or edit. All commands that start with a colon (:) are ex commands. A typical list includes a set command, and possibly a few map commands. Put these commands on one line by separating them with the pipe (|) character. If you use the c shell, *csh*, put a line like this in the

```
setenv EXINIT 'set ai aw terse|map @ dd|map # x'
```

This sets the options autoindent, autowrite, terse, (the set *command), and makes @ delete a line, (the first* map*), and makes # delete a character, (the second* map*). (See the "Macros" section for a description of the* map *command.)*

If you use the Bourne shell, put these lines in the file *.profile* in your home directory:

```
EXINIT='set ai aw terse|map @ dd|map # x'
export EXINIT
```

Of course, the particulars of the line depend on the options you want to set.

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, vi saves the last nine deleted blocks of text in a set of numbered registers 1–9. You can get the nth previous deleted text back in your file by "np. 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, that *puts* text in the buffer after the cursor. If this doesn't bring back the text you

wanted, type u to undo this and then (period) . to repeat the p. In general the
'.' command repeats 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 that 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 recovered text. You can
also use P rather than p to put the recovered text before rather than after the cur-
sor.

**Recovering Lost Files — the
−r Option**

If something goes wrong so the system goes down, you can recover the work you
were doing up to the last few changes. You will normally receive mail when you
next log in giving you the name of the file that has been saved for you. You
should then change to the directory where you were when the system went down
and type:

```
hostname% vi −r filename
```

replacing *filename* with the name of the file 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. vi 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 that you changed. You can either choose to discard the
changes you made (if they are easy to redo) or to replace the few lost lines by
hand.

You can get a listing of the files that are saved for you by typing:

```
hostname% vi −r
```

If there is more than one instance of a particular file saved, vi 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.

The invocation 'vi −r' will not always list all saved files, but they can be
recovered even if they are not listed.

**Continuous Text Input —
wrapmargin**

When you are typing in large amounts of text it is convenient to have lines bro-
ken near the right margin automatically. To do this, use the *set wrapmargin*
option:

```
:se wm=10<CR>
```

This rewrites words on the next line that you type past the right margin.

If vi 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. vi supplies blank space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this blank space. You can delete the blank space with x if you don't want it.

If you want to *split* a line into two, put the cursor where you want the break, and type rCR.

Features for Editing Programs

vi has a number of commands for editing programs. To generate correctly-indented programs, use the *autoindent* option:

```
:se ai<CR>
```

Now try opening a new line with o. Type a few tabs on the line and then some characters. If you type a CR and start another line, notice that vi supplies blank space at the beginning of the line to align the text of the new line with that of the previous line.

After you have started a new line, you might want to indent your current line less than the previous line. You are still in insert mode, and cannot backspace over the automatic indentation. However, you can type CTRL-D to backtab over each level of indentation. Each time you type CTRL-D, you back up one position, normally to an eight-column boundary. You can set the number of columns that a tab shifts with the *shiftwidth* option. 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 text 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 type %. This shows you the matching parenthesis. This works also for braces { and }, and brackets [and].

If you are editing C programs, you can use [[and]] to advance or retreat to a line starting with a {, that is, a function declaration at a time. When you use]] with an operator, it stops after a line that starts with }; this is sometimes useful with y]].

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 that the blank line ends a paragraph. The result is sorted text in your file.

Commands for Editing LISP

If you are editing a LISP program, 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. Use { and } to skip to the next list, or through a comment quickly.

The *autoindent* option works differently for LISP, supplying indentation to align at the first argument to the last open list. If there is no such argument, the indent is two spaces more than the last level.

The *showmatch* option shows matching parentheses. Try setting it with:

```
:se sm<CR>
```

and then try typing a '(' *some words* and then a ')'. Notice that the cursor briefly shows the position of the '(' which matches the ')'. This happens only if the matching '(' is on the screen, and the cursor stays there for at most one second.

vi 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 realigns 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.

Macros

vi has a parameterless macro facility you can set up so that when you type a single keystroke, vi will act as though you had typed some longer sequence of keys. Set this up if you find yourself repeatedly typing the same sequence of commands or text. There are two kinds of macros:

1. 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.

2. You can use the map command from vi (typically in your EXINIT) with a command of the form:

```
:map lhs rhs<CR>7
```

This maps *rhs* into *lhs*. There are restrictions: *lhs* should be one keystroke (either

---

[7] *lhs* is an abbreviation for *left hand side*. *rhs* is an abbreviation for *right hand side*.

one character or one function key) since it must be entered within one second unless *notimeout* (see the "Option Descriptions" section) is set. In that case you can type it as slowly as you wish, and vi will wait for you to finish before it echoes anything). The *lhs* can be no longer than ten characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs*, escape them with a CTRL-V. It may be necessary to double the CTRL-V if you use the map command inside vi, rather than in ex. You do not need to escape spaces and tabs inside the *rhs*.

Thus to make the q key write and exit vi, type:

```
:map q :wq^V<CR><CR>
```

which means that whenever you type q, it will be as though you had typed the four characters :wqCR. A CTRL-V is needed to quote the first CR; the second CR ends the *map* definition. Actually the first CR is part of the *rhs*, while the second terminates the : command.

You can delete macros with

```
:unmap lhs
```

If the *lhs* of a macro is '#0' through '#9', this maps the particular function key instead of the two-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. You can change the character '#' 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 applies the mapping to input mode, rather than command mode. So, to arrange for CTRL-T to be the same as four spaces in input mode, type:

```
:map! ^T ^Vb/b/b/b/
```

where b represents a blank. The CTRL-V prevents the blanks from being taken as blank space between the *lhs* and *rhs*. Type simply:

```
:map!
```

to list macros that apply during input mode and

```
:map
```

to list macros that apply during command mode.

Word Abbreviations — `:ab`
`:una`

A feature similar to macros in input mode is word abbreviation. You can type a short word and have it expanded into a longer word or words with `:abbreviate` (`:ab`). For example:

```
:ab foo find outer otter
```

always changes the word 'foo' into the phrase 'find outer otter'. Word abbreviation is different from macros in that only whole words are affected. If 'foo' 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. This only operates in visual mode and uses the same syntax as the `map` command, except that there are no '!' forms.

Use `:unabbreviate` (`:una`) to turn off the abbreviation. To unabbreviate the above, for example, type:

```
:una foo
```

The `vi` editor has a number of short commands that abbreviate the longer commands we have introduced here. You can find these commands easily in the "`ex` Commands" section of the "`ex` Quick Reference." They often save a bit of typing, and you can learn them when it's convenient.

2.8. Nitty-gritty Details

The following presents some functional details and some `ex` commands (see the "File Manipulation Commands" section) that are particularly useful in `vi`.

Line Representation in the Display

`vi` folds long logical lines onto many physical lines in the display. Commands that advance lines advance logical lines and 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 over 80 columns long. You can make long lines very easily by placing the cursor on the first line of two you want to join and typing shift-J (capital J).

`vi` only puts full lines on the display; if there is not enough room on the display to fit a logical line, the `vi` editor leaves the physical line empty, placing only an '@' on the line as a place holder. (When you delete lines on a dumb terminal, `vi` 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 with CTRL-R.

If you wish, you can have the editor place line numbers before each line on the display. To enable this, type the option:

```
:se nu<CR>
```

sun
microsystems

To turn it off, use the *no numbers* option:

```
:se nonu<CR>
```

You can have tabs represented as CTRL-I (appears as ^I) and the ends of lines indicated with '$' by giving the *list* option:

```
:se list<CR>
```

To turn this off, use:

```
:se nolist<CR>
```

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 that are past the logical end of file.

Command Counts

Most vi commands use a preceding count to affect their behavior in some way. The following table lists the common ways the counts are used:

New window size	:	/	?	[[	]]	`	'
Scroll amount	CTRL-D	CTRL-U					
Line/column number	z	G	|				
Repeat effect	Most of the rest						

vi maintains a notion of the current default window size. (On terminals that run at speeds greater than 1200 baud, vi uses the full terminal screen. On terminals slower than 1200 baud, and most dialup lines are in this group, vi uses eight lines as the default window size. At 1200 baud, the default is 16 lines.)

vi uses the default window size when it clears and refills the screen after a search or other motion moves far from the edge of the current window. All commands that take a new window size as count often redraw the screen. 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 CTRL-D. The window will revert to the last specified size the next time it is cleared and refilled, but not by a CTRL-L which just redraws the screen as it is.

The scroll commands CTRL-D and CTRL-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 inserts ten repetitions of a plus sign followed by four minus signs:

```
+----+----+----+----+----+----+----+----+----+----
```

A few commands also use a preceding count as a line or column number.

Except for the few commands that ignore any counts, such as CTRL-R, the rest of the vi commands use a count to indicate a simple repetition of their effect. Thus 5w advances five words on the current line, while 5RETURN 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 delete first one and then three words. You can then delete two more words with 2..

File Manipulation Commands

The following table lists the file manipulation commands you can use when you are in vi.

Table 2-2   *File Manipulation Commands*

Command	Meaning
:w	Write back changes
:wq	Write and quit
:x	Write (if necessary) and quit (same as ZZ).
:e *name*	Edit file *name*
:e!	Re-edit, 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*

A CR or ESC follows all of these commands. The most basic commands are :w and :e. End a normal editing session on a single file with a ZZ command. If you are editing for a long period of time, use the :w command occasionally after major amounts of editing, and then finish with a ZZ. When you edit more than one file, you can finish with one 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, give an ! after the command you would otherwise use to exit without changing the file. Use this carefully.

Use the :e command with 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, usually a scan like +/*pat* or +?*pat*. In forming new names to the e command, use the character % which is replaced by the current filename, or the character # which is replaced by the alternate filename. The alternate filename 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 CTRL-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 filename.

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. To respecify the list of files to be edited, give the :n command a list of filenames, or a pattern to be expanded as you would have given it on the initial vi command.

For editing large programs, use the :ta command. It uses a database of function names and their locations, which can be created by programs such as ctags(1) to quickly find a function whose name you give; see the *SunOS Reference Manual* for details. If the :ta command requires 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.

More about Searching for Strings

When you are searching for strings in the file with / and ?, vi 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, vi will affect characters up to the match place, rather than whole lines; thus use +0 to affect the line that matches.

To have vi ignore the case of words in searches, give the *ignorecase* option:

```
:se ic<CR>
```

To turn this off so that vi recognizes case again, use:

```
:se noic<CR>
```

Strings given to searches may actually be regular expressions. If you do not want or need this facility, you should put:

```
set nomagic
```

in your EXINIT. When nomagic is set, only the characters caret (^) and dollar sign ($) are special in patterns. The character backslash (\) is also special with nomagic set. You can precede some of the normally special characters (not special in nomagic mode) with a backslash to enable their special properties.

It is necessary to use a backslash (\) before a slash (/) to search for a slash character in a forward scan and before a question mark (?) to search for a question mark in a backward scan. The command to search for a slash character is shown on the last line of the example below, as it would appear on your screen.

```
text text text text text text text text text text text
text text text/text text text text text text text
text text text text text text text text text text
text text? text text text text text text text text? text
text text text/text text text text text/text text
text text text text text text text text text text
//<CR>
```

The following table gives the extended forms when `magic` is set.

Table 2-3   *Extended Pattern Matching Characters*

Character	Meaning
^	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
[*string*]	Matches any single character in *string*
[^*string*]	Matches any single character not in *string*
[*x–y*]	Matches any character between *x* and *y*
*	Matches any number of the preceding pattern

If you use *nomagic* mode, use the ' . [' and '*' primitives with a preceding \

More about Input Mode

There are a number of characters to make corrections during input mode. These are summarized in the following table.

Table 2-4   *Input Mode Corrections*

Character	Meaning
CTRL-H	Deletes the last input character
CTRL-W	Deletes the last input word
erase	Your erase character, same as CTRL-H
kill	Your kill character, deletes the input on this line
\	Escapes a following CTRL-H and your erase and kill
ESC	Ends an insertion
DEL	Interrupts an insertion, terminating it abnormally
CR	Starts a new line
CTRL-D	Backtabs over *autoindent*
0CTRL-D	Kills all the *autoindent*
^CTRL-D	Same as CTRL-D, but restores indent next line
CTRL-V	Quotes the next non-printing character into the file

The most usual way of making corrections to input is to type DEL (CTRL-H on a

terminal) to correct a single character, or by typing one or more CTRL-W to back over incorrect words.

Your system kill character CTRL-U (or sometimes CTRL-X) erases 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 you did not insert with this insertion command. To make corrections on the previous line after a new line has been started, press ESC to end the insertion, move over and make the correction, and then return to where you were to continue. Use A to append at the end of the current line; this is often useful for continuing text input.

If you wish to type in your erase or kill character, say CTRL-U, 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 CTRL-V. The CTRL-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.[8]

If you are using *autoindent*, you can backtab over the indent that it supplies by typing a CTRL-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 a caret (CTRL-) and then CTRL-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 zero (0) followed immediately by a CTRL-D if you wish to kill all the indent and not have it come back on the next line.

2.9. Command and Function Reference

The following section provides abridged explanations of the various vi and ex commands.

Notation

Notation used in this section is as follows.

[option]	Denotes optional parts of a command. Many vi commands have an optional count, explained below.
[count]	Means that an optional number may precede the command to multiply or iterate the command.
{variable item}	Denotes parts of the command that must appear, but can take any 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 type the ESCAPE key. <a−z> means type a lower-case letter. CTRL-*<character>* means type the character as a *control* character, that is, with the CTRL key held down while

---

[8] This is not quite true. vi does not allow the NULL (CTRL-@) character to appear in files. Also, the editor uses LF (linefeed or CTRL-J) to separate lines in the file, so it cannot appear in the middle of a line. You can insert most non-printing characters, however, after a CTRL-V. The exceptions are CTRL-S or CTRL-Q (for suspending and resuming output), which vi ignores; use ex if you need to insert these characters.

simultaneously typing the specified character. Here we indicate control characters with *upper-case* letters, but CTRL-<uppercase letter> and CTRL-<lower-case letter> are equivalent. That is, CTRL-D is equal to CTRL-d. The most common character abbreviations used in this list are as follows:

Table 2-5   *Common Character Abbreviations*

Character Abbreviation	Meaning	Hexadecimal Representation
ESC	escape	0x1b
CR	carriage return, CTRL-M	0xd
<lf>	linefeed CTRL-J	0xa
<nl>	newline, CTRL-J	0xa (same as linefeed)
<bs>	backspace, CTRL-H	0x8
<tab>	tab, CTRL-I	0x9
<bell>	bell, CTRL-G	0x7
<ff>	formfeed, CTRL-L	0xc
<sp>	space	0x20
DEL	delete	0x7f

Commands

Following are brief explanations of the vi commands categorized by function for easy reference.

Entry and Exit

To use vi to edit a particular file, type:

```
hostname% vi filename
```

vi will read the file intro the buffer, and place the cursor at the beginning of the first line. The first screenful of the file is displayed on the screen.

To exit from vi, type:

```
ZZ   (or :x or :q or :q!)
```

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.

Cursor and Page Motion

Note: You can move the cursor on your screen with the arrow keys on your workstation keyboard, the control character versions, or the h, j, k, and l keys. If you are using a terminal that does not have arrow keys, use the control character versions or the h, j, k, and l keys.

[count]<bs> or [count]h or [count]←

Move the cursor to the left one character. Cursor stops at the left margin of the page. [count] specifies the number of spaces to move.

[count]<lf> or [count]j or [count]↓

Also [count]CTRL-N. Move the cursor down one line. Moving off the screen scrolls the window to force a new line onto the screen. Mnemonic: Next.

[count]k or [count]↑	Also [count]CTRL-P. Move the cursor up one line. Moving off the top of the screen forces new text onto the screen. Mnemonic: Previous.
[count]<sp> or [count]l or [count]→	Move the cursor right one character. Cursor will not go beyond the end of the line.
[count]−	Move the cursor up the screen to the beginning of the next line. Scroll if necessary.
[count]+ or [count]CR	Move the cursor down the screen to the beginning of the next line. Scroll up if necessary.
[count]$	Move the cursor to the end of the line. If there is a count, move to the end of the line *count* 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.
[count]l	Move the cursor to the column specified by the count. The default is column zero.
[count]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-word
[count]W	Move the cursor to the beginning of the next word that follows a blank space (<sp>,<tab>, or <nl>). Ignore other punctuation.
[count]b	Move the cursor to the preceding word. Mnemonic: backup-word
[count]B	Move the cursor to the preceding word that is separated from the current word by a blank space (<sp>,<tab>, or <nl>).
[count]e	Move the cursor to the end of the current word or the end of the *count*th word hence. Mnemonic: end-of-word
[count]E	Move the cursor to the end of the current word which is delimited by blank 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: Go-to
	Note: The next four commands (CTRL-D, CTRL-U, CTRL-F, CTRL-B) are not true motion commands, in that they cannot be used as the object of commands such as delete or change.
[count]CTRL-D	Move the cursor down in the file by *count* lines (or the last *count* if a new count isn't given). The initial default is half a page. The screen is simultaneously scrolled up. Mnemonic: Down
[count]CTRL-U	Move the cursor up in the file by *count* lines. The screen is simultaneously scrolled down. Mnemonic: Up
[count]CTRL-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: Forward

☀ sun
microsystems

[count]CTRL-B	Move the cursor to the previous page. Two lines of the current page are kept if possible. Mnemonic: Backward
[count])	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>.
[count](	Move the cursor backward to the beginning of a sentence.
[count]}	Move the cursor to the beginning of the next paragraph. This command works best inside nroff documents. It understands the nroff macros in –ms, for which the commands .IP, .LP, .PP, .QP, 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 are adjustable. See the entry for "Paragraphs" in the "Set Commands" section.
[count]{	Move the cursor backward to the beginning of a paragraph.
]]	Move the cursor to the next 'section,' where a section is defined by the set of nroff macros in –ms, 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 the "Sections" entry under the heading "Set Commands."
[[	Move the cursor backward 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 }, it 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 on that line until it finds one and then jumps to the match mate.
[count]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 *count* lines from the top of the screen. Mnemonic: Home
[count]L	If there is no count, move the cursor to the beginning of the last line on the screen. If there is a count, move the cursor to the beginning of the line *count* lines from the bottom of the screen. Mnemonic: Last
M	Move the cursor to the beginning of the middle line on the screen. Mnemonic: Middle
m<a-z>	*Mark* the place in the file without moving the cursor; use a character from a to z, '<a-z>', as the label for referring to this location in the file. See the next two commands. Mnemonic: mark 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 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 searching or jumping 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.

Searches   The following commands search for items in a file.

[count]f{chr}   Search forward on the line for the next or *count*th occurrence of the character *chr*. The cursor is placed *at* the character of interest. Mnemonic: find character

[count]F{chr}   Search backward on the line for the next or *count*th occurrence of the character *chr*. The cursor is placed *at* the character of interest.

[count]t{chr}   Search forward on the line for the next or *count*th occurrence of the character *chr*. The cursor is placed *just preceding* the character of interest. Mnemonic: move cursor up to character

[count]T{chr}   Search backward on the line for the next or *count*th occurrence of the character *chr*. The cursor is placed *just preceding* the character of interest.

[count];   Repeat the last f, F, t or T command in the same search direction.

[count],   Repeat the last f, F, t or T command, but in the opposite search direction. This is useful if you overshoot what you are looking for.

[count]/[string]/<nl>   Search forward for the next occurrence of 'string'. Wraparound at the end of the file does occur. The final / is not required.

[count]?[string]?<nl>   Search backward for the next occurrence of 'string'. If a count is specified, the count becomes the new window size. Wraparound at the beginning of the file does occur. The final ? is not required.

n   Repeat the last /[string]/ or ?[string]? search. Mnemonic: next 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 like you can in the ed editor.

Text Insertion   The following commands insert text. Terminate all multi-character text insertions with an ESC character. You can always *undo* the last change by typing a u. The text insert in insertion mode can contain newlines.

a{text}<esc>   Insert text immediately following the cursor position. Mnemonic: append

A{text}<esc>   Insert text at the end of the current line. Mnemonic: Append

i{text}<esc>   Insert text immediately preceding the cursor position. Mnemonic: insert

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: open new line

O{text}<esc>	Insert a new line preceding the line on which the cursor appears and insert text there.

Text Deletion

The following commands delete text in various ways. You can always *undo* changes by typing the u command.

[count]x	Delete the character or characters starting at the cursor position.
[count]X	Delete the character or characters starting at the character preceding the cursor position.
D	Delete the remainder of the line starting at the cursor. Mnemonic: Delete the rest of line
[count]d{motion}	Delete one or more occurrences of the specified motion. You can use any motion here described in the sections "Low Level Character Motions" and "Higher Level Text Objects." You can repeat the d (such as [*count*]dd) to delete *count* lines.

Text Replacement

Use the following commands to simultaneously delete and insert new text. You can *undo* all such actions by typing u following the command.

r<chr>	Replace the character at the current cursor position with <chr>. This is a one-character replacement. No ESC is required for termination. Mnemonic: replace character
R{text}<esc>	Start overlaying the characters on the screen with whatever you type. It does not stop until you type an ESC.
[count]s{text}<esc>	Substitute for *count* characters beginning at the current cursor position. A '$' appears at the position in the text where the *count*th character appears so you will know how much you are erasing. Mnemonic: substitute
[count]S{text}<esc>	Substitute for the entire current line or lines. If you do not give a count, a '$' appears at the end of the current line. If you give a count of more than 1, all the lines to be replaced are deleted before the insertion begins.
[count]c{motion}{text}<esc>	Change the specified *motion* by replacing it with the insertion text. A '$' appears at the end of the last item that is being deleted unless the deletion involves whole lines. Motions can be any motion from the sections "Low Level Character Motions" and "Higher Level Text Objects." Repeat the c (such as [*count*]cc) to change *count* lines.

Moving Text

You can move chunks of text around in a number of ways with vi. There are nine buffers into which each piece of text 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 you precede any delete or replacement type command by "<a-z>, that named buffer will contain the text deleted after the command is executed. For example, "a3dd deletes three lines starting at the current line and puts them in buffer "a. 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. There are two more basic commands and some variations useful in getting and putting text into a file.

["<a-z>][count]y{motion}   Yank the specified item or *count* items and put in the *undo* buffer or the specified buffer. The variety of *items* that you can yank is the same as those that you can delete 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>][count]Y   Yank the current line or the *count* 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: Yank

["<a-z>]p   Put *undo* buffer or the specified buffer down *after* the cursor. If you yanked or deleted whole lines into the buffer, they are put down on the line following the line the cursor is on. If you deleted something else, like a word or sentence, it is inserted immediately following the cursor. Mnemonic: put buffer

Note that text in the named buffers remains there when you start editing a new file with the : e *fileCR* 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 you yanked or deleted whole lines into the buffer, they are put down on the line preceding the line the cursor is on. If you deleted something else, like a word or sentence, it is inserted immediately preceding the cursor.

[count]>{motion}   The shift operator right shifts 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 the "Terminal Information" section.) >> means right shift the current line or lines.

[count]<{motion}   The shift operator left shifts 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 the section on "Terminal Information.") << means left shift the current line or lines. Once the line has reached the left margin, it is not affected further.

[count]={motion}   Prettyprints the indicated area according to LISP conventions. The area should be a LISP s-expression.

Miscellaneous Commands   A number of useful miscellaneous vi commands follow:

ZZ   Exit from vi. If any changes have been made, the file is written out. Then you are returned to the shell.

CTRL-L   Redraw the current screen. This is useful if messages from a background process are displayed on the screen, if someone 'writes' to you while you are using vi or if for any reason garbage gets onto the screen.

CTRL-R   On dumb terminals, those not having the 'delete line' function (the vt100 for example), 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 want to

	actually get rid of the lines marked with '@' and see what the page looks like, type a CTRL-R.
CTRL-T	An abbreviation synonym for the *pop* command, used to manipulate the tagstack.
~	Converts the character under the cursor to lower-case if it is upper-case, or to upper-case if it is lower-case. With a count, converts the specified number of characters at and to the right of the cursor. The cursor ends up after the last character converted.
.	'Dot' repeats the last text modifying command. You can type a command once and then move to another place and repeat it by just typing '.'.
u	Undo the last command that changed the buffer. Perhaps the most important command in the editor. Mnemonic: undo
U	Undo all the text modifying commands performed on the current line since the last time you moved onto it.
[count]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', two spaces are inserted. A count joins the next *count* lines. Mnemonic: Join lines
Q	Switch to ex editing mode. In this mode vi behaves very much like ed — it operates on single lines and does not attempt to keep the window up to date. Once in this mode you can also switch to the *open* mode of editing by entering the command [*line number*] open<nl>, which is similar to normal visual mode except the window is only one line long. Mnemonic: Quit visual mode
CTRL-]	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.
[count]!{motion}{Sun cmd}<nl>	Any Sun system filter (that is, a 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 you can 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 Sun system command. Note: To escape to the shell for just one command, use :!{cmd}<nl> (see the "High Level Commands" section).
z{count}<nl>	Reset the current window size to *count* lines and redraw the screen.

Special Insert Characters

Following are some characters that have special meanings during insert mode.

CTRL-V	During inserts, typing a CTRL-V quotes control characters into the file. Any character typed after the CTRL-V is inserted into the file.
[^]CTRL-D	CTRL-D without any argument backs up one *shiftwidth*. Use this to remove indentation that was inserted by the *autoindent* feature. Typing ^CTRL-D temporarily removes all the autoindentation, thus placing the cursor at the left margin. On the next line, the previous indent level is restored. This is useful for

putting 'labels' at the left margin. 0CTRL-D removes all autoindents and keeps it that way. Thus the cursor moves to the left margin and stays there on successive lines until you type TABs. As with the TAB, the CTRL-D is effective only before you type any other 'non-autoindent' controlling characters. Mnemonic: Delete a shiftwidth

CTRL-W   If the cursor is sitting on a word, CTRL-W moves the cursor back to the beginning of the word, erasing the word from the insert. Mnemonic: erase Word

<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, quote it with the CTRL-V.

: Commands

Typing a colon (:) during command mode puts the cursor at the bottom on the screen in preparation for a command. In the : mode, you can give vi most ex commands. You can also exit from vi or switch to different files from this mode. Terminate all commands of this variety by a <nl>, <cr>, or ESC.

:w[!] [file]   Write out the current text to the disk. It is written to the file you are editing unless you supply *file*. If *file* is supplied, the write is directed to that file instead. If that file already exists, vi does not write unless you use the '!' indicating you *really* want to write over the older copy of the file.

:q[!]   Exit from vi. If you have modified the file you are currently looking at and haven't written it out, vi refuses to exit unless you type the !.

:e[!] [+[cmd]] [file]   Start editing a new file called *filename* 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'. Use it if you really mess up the file. The optional '+' says instead of starting at the beginning, start at the 'end', or, if you supply *cmd*, execute *cmd* first. Use this where *cmd* is *n* (any integer) that starts at line number *n*, and /*text* searches for 'text' and starts at the line where it is found.

CTRL-^   Switch back to the place in the previous file that you were editing with vi, before you switched to the current file. (Same as :e # on the command line.)

:n[!]   Start editing the next file in the argument list. Since you can call vi with multiple filenames, the :n command tells it to stop work on the current file and switch to the next file. If you have modified the current file, it has to be written out before the :n will work or else you must use '!', which discards the changes you 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 system 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. If the current file is up to date on the disk, it switches to the file specified and uses the search pattern specified to find the 'tagged' item of interest. Use this when editing multi-file C programs such as the operating system. There is a program called `ctags` which generates an appropriate `tags` file for C and f77 programs, so that by saying `:ta` *function* you can switch to that function. It can also be useful when editing multi-file documents, though the `tags` file has to be generated manually in this case.

:pop   You can pop tags from the stack either with the `:pop` directive, or with the CTRL-T command. The directive `:set notagstack` eliminates this stack altogether.

Set Commands

`vi` has a number of internal variables and switches you can set to achieve special affects. These options come in three forms: switches that toggle off or on, options that require a numeric value, and options that require an alphanumeric string value. Set the toggle options by a command of the form:

```
:set option<nl>
```

and turn off the toggle options with the command:

```
:set nooption<nl>
```

To set commands requiring a value, use 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 described in the following list as well as the normal default value.

To use values other than the default every time you enter `vi`, place the appropriate *set* command in your EXINIT environment, such as (for the C shell):

```
setenv EXINIT 'set ai aw terse sh=/bin/csh'
```

Or (for the Bourne shell):

```
EXINIT='set ai aw terse sh=/bin/csh'
export EXINIT
```

Place these in your `.login` or `.profile` file 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 CTRL-T moves this boundary to the right; to move it to the left, use CTRL-D.

autoprint ap
Default: ap Type: toggle
Displays the current line after each `ex` text modifying command. Not of much interest in the normal `vi` visual mode.

autowrite aw
Default: noaw type: toggle
Does an automatic write if there are unsaved changes before certain commands that change files or otherwise interact with the outside world are executed. These commands are `:!`, `:tag`, `:next`, `:rewind`, CTRL-^, and CTRL-].

beautify bf
Default: nobf Type: toggle
Discards all control characters except <tab>, <nl>, and <ff>.

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>. Sun Workstations not equipped with speakers flash instead of beeping.

flash fl
Default: fl Type: toggle
If the terminal has visual bell capability, use that instead of the audible 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 Sun system.

ignorecase ic
Default: noic Type: toggle
Map all upper-case characters 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
Show the <tab> and <nl> characters visually on all displayed lines.

magic
Default: magic Type: toggle
Enable the metacharacters for matching. These include `.`, `*`, `<`, `>`, `[`*string*`]`, `[^`*string*`]`, and `[<`*chr*`>-<`*chr*`>]`.

number nu	Default: nonu Type: toggle Display each line with its line number.
optimize opt	Default: opt Type: toggle Useful 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 type-ahead on some versions of the operating system.
paragraphs para	Default: para=IPLPPPQPP LIpplpipnpbp Type: string Each pair of characters in the string indicates nroff macros to be treated as the beginning of a paragraph for the { and } commands. The default string is for the −ms macros. To indicate one-letter nroff macros, such as .P or .H, insert a space for the second character position. For example:

```
:set paragraphs=PPH\ bp<nl>
```

causes vi to consider .PP, .H and .bp as paragraph delimiters.

prompt	Default: prompt Type: toggle In ex command mode the prompt character : is displayed 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 Set the threshold for the number of lines modified. When more than this number of lines is modified, removed, or yanked, vi reports 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 CTRL-U and CTRL-D commands.
sections	Default: sections=NHSHH HUuhsh+c Type: string Each two-character pair of this string specifies nroff macro names that are to be treated as the beginning of a section by the]] and [[commands. The default string is for the −ms macros. To enter one-letter nroff macros, use a quoted space as the second character. See the "Paragraphs" entry for a fuller explanation.
shell sh	Default: sh=from environment SHELL or /bin/sh Type: string Specify the name of the *sh* to be used for 'escaped' commands.
shiftwidth sw	Default: sw=8 Type: numeric Specify the number of spaces that a CTRL-T or CTRL-D will move over for indenting, and the amount that < and > will 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.

showmode smd   Default: nosmd Type: toggle
Who the current input mode on the message line:

state	display
command mode	(nothing)
in a command	APPEND MODE
in s command	SUBSTITUTE MODE
in c command	CHANGE MODE
in R command	REPLACE MODE
in o command	OPEN MODE
in i command	INSERT MODE
in r command	REPLACE 1 CHAR

slowopen slow   Default: terminal dependent Type: toggle
Prevent updating the screen some of the time to improve speed on terminals that are slow and dumb.

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 type a Q to change to command mode, type a set term command, and enter vi to get back into visual. Or exit from vi, fix $TERM, and re-enter. The definitions that drive a particular terminal type are in the file /etc/termcap.

terse   Default: terse Type: toggle
When set, the error diagnostics are short.

warn   Default: warn Type: toggle
Warns if you try 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
Specify 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   Set the 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
```

produces a four-line window at speeds up to 600 baud, a 12-line window at 1200 baud, and a full-screen window (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 automatically inserts 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 means that any time you are within 10 spaces of the right margin, vi looks for a <sp> or <tab> that it can replace with a <nl>. This is convenient if you forget to look at the screen while you type. If you go past the margin (even in the middle of a word), the entire word is erased and rewritten on the next line.

writeany wa	Default: nowa Type: toggle

vi normally makes a number of checks before it writes out a file. This prevents you from inadvertently destroying a file. When the *writeany* option is enabled, vi no longer makes these checks.

Character Functions

This section describes how the editor uses each character. The characters are presented in their order in the ASCII character set: control characters come first, then most special characters, the digits, upper-, and finally lower-case characters. For each character we list its meaning as a command and its meaning (if any) during insert mode.

CTRL-@   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 CTRL-@ cannot be part of the file due to the editor implementation.

CTRL-A   Unused.

CTRL-B   Scroll backward one window. A count specifies repetition. The top two lines in the window before typing CTRL-B appear as the bottom two lines of the next window.

CTRL-C   Unused.

CTRL-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 CTRL-D and CTRL-U commands. During an insert, CTRL-D backtabs over *autoindent* blank space at the beginning of a line. This blank space cannot be backspaced over.

CTRL-E   Exposes one more line below the current screen in the file, leaving the cursor where it is if possible.

CTRL-F   Move forward one window. A count specifies repetition. The bottom two lines in the window before typing CTRL-F appear as the top two lines of the next window.

CTRL-G   Equivalent to : fCR. These commands display the current file, a message if the file has been modified, the line number of the line the cursor is on, the total number of lines in the file, and the percentage of the way through the file that the current line is.

CTRL-H (BS)   Same as ← (see h). During an insert, CTRL-H eliminates the last input character, backing over it but not erasing it; the character remains so you can see what you typed if you wish to type something only slightly different.

CTRL-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 that represent the tab. The *tabstop* option controls the spacing of tabstops.
CTRL-J (LF)	Same as ↓ (see j).
CTRL-K	Unused.
CTRL-L	The ASCII formfeed character, that clears and redraws the screen. 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.
CTRL-M (CR)	A carriage return advances to the next line, at the first non-blank position in the line. Given a count, it advances that many lines. During an insert, a CR causes the insert to continue onto another line.
CTRL-N	Same as ↓ (see j).
CTRL-O	Unused.
CTRL-P	Same as ↑ (see k).
CTRL-Q	Not a command character. In input mode, CTRL-Q quotes the next character, the same as CTRL-V , except that some teletype drivers will eat the CTRL-Q so that vi never sees it. Resumes operation suspended by CTRL-S.
CTRL-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.
CTRL-S	Some teletype drivers use CTRL-S to suspend output until CTRL-Q is pressed. Unused.
CTRL-T	Not a command character. During an insert with *autoindent* set and at the beginning of the line, inserts *shiftwidth* blank space.
CTRL-U	Scrolls the screen up half a window, the reverse of CTRL-D, which scrolls down. Counts work as they do for CTRL-D, and the previous scroll amount is common to both CTRL-D and CTRL-U. On a dumb terminal, CTRL-U will often necessitate clearing and redrawing the screen further back in the file.
CTRL-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.
CTRL-W	Not a command character. During an insert, backs up as b does in command mode; the deleted characters remain on the display (see CTRL-H).
CTRL-X	Unused.
CTRL-Y	Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; CTRL-Y is the reverse of CTRL-E).
CTRL-Z	Stops the editor, exiting to the top level shell. Same as : stopCR.
CTRL-[(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 flashes the screen or rings the bell. You

can thus type ESC if you don't know what is happening till the editor flashes the screen. If you don't know if you are in insert mode, you can type ESCa, and then material to be input; the material is inserted correctly whether or not you were in insert mode when you started.

CTRL-\   Unused.

CTRL-]   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'.

CTRL-^   Equivalent to `:e #CR`, returning to the previous position in the last-edited file, or editing a file that you specified if you got a `No write since last change diagnostic` and do not want to have to type the filename again. You have to do a `:w` before CTRL-^ will work in this case. If you do not wish to write the file you should do `:e! #CR` instead.

CTRL-_   Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.

SPACE   Same as → (see l).

!   An operator that 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 filters the count lines; otherwise the count is passed on to the object after the !. Thus `2!}fmtCR` reformats the next two paragraphs by running them through the program `fmt`. If you are working on LISP, the command `!%grindCR`, given at the beginning of a function, will run the text of the function through the LISP grinder. (The `grind` command may not be present at all installations.) To read the output of a command into the buffer, use `:rcommand`. To simply execute a command, [from `vi`], use `:!command`.

"   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.

The macro character which, when followed by a number, will substitute for a function key on terminals without function keys. 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 listCR`, the end of each line is indicated by showing 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 next line.

%   Moves to the parenthesis or brace { } that 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 that was marked with this letter with a m command, at the first non-blank character in the line.

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 ? and 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 the " { " and " [[" entries below). A count advances that many sentences.

)   Advances to the beginning of a sentence. A count repeats the effect. See (above for the definition of a sentence.

*   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 typing too many ; characters. A count repeats the search.

—   Retreats to the previous line at the first non-blank 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 scrolling is not possible. If a large amount of scrolling is required, the screen is also cleared and redrawn, with the current line at the center.

.   Repeats the last command that changed the buffer. Especially useful when deleting words or lines; you can delete some words or lines and then type . to delete more words or lines. Given a count, it passes it on to the command being repeated. Thus after a 2dw, 3. deletes three words.

/   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 ESC returns to command state without ever searching. The search begins when you type CR to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress; you can then terminate the search with a CTRL-C (or 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 affect whole lines. To do this, give a pattern with 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 may speed 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. Unless you set nomagic in your *.login* file (*?*), you will have to precede the characters ., [, *, and ~ in the search pattern with a \ to get them to work as you would naively expect.

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.

:   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 a CR, and the command is then executed. You can return to where you were by typing ESC or DEL if you type : accidentally.

;   Repeats the last single character find that used f, F, t, or T. A count iterates the basic scan.

<   An operator that 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.

=   Reindents line for LISP, as though they were typed in with *lisp* and *autoindent* set.

>   An operator that shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in >>. Counts repeat the basic object.

?   Scans backward, the opposite of /. See the / description above for details on scanning.

@   A macro character. 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.

A   Appends at the end of line; a synonym for $a.

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.

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 backward in the current line. A count repeats this search that many times.

G   Goes to the line number given as preceding argument, or to the end of the file if you do not give a preceding count. The screen is redrawn with the new current line in the center if necessary.

H   Home arrow. Homes the cursor to the top line on the screen. If a count is given, the cursor is moved to the count'th line on the screen. In any case the cursor is moved to the first non-blank character on the line. If used as the target of an operator, full lines are affected.

I   Inserts at the beginning of a line; this is equivalent to '0i'.

J   Joins together lines, supplying appropriate blank 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.

K   Unused.

L   Moves the cursor to the first non-blank character of the last line on the screen. With a count, to the first non-blank of the count'th line from the bottom. Operators affect whole lines when used with L.

M   Moves the cursor to the middle line on the screen, at the first non-blank position on the line.

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.

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.

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.

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.

U   Restores the current line to its state before you started changing it.

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.

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.

ZZ   Exits the editor. (Same as :xCR.) 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 that start with a formfeed CTRL-L. Lines beginning with { also stop [[; this makes it useful for looking backward, a function at a time, in C programs. If the *lisp* option is set, stops at each (at the beginning of a line, and is thus useful for moving backward at the top level LISP objects.

\   Unused.

]] Forward to a section boundary; see [[for a definition.

^   Moves to the first non-blank position on the current line.

_   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 that was marked with this letter with an 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.

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. Terminate the insertion with an ESC.

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.

c   An operator that 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 that is changed is saved in the numeric named buffers. If only part of the current line is affected, 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.

d   An operator that 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.

e   Advances to the end of the next word, defined as for b and w. A count repeats the effect.

f   Finds the first instance of the next character following the cursor on the current line. A count repeats the find.

g   Unused.

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, CTRL-H has the same effect. A count repeats the effect.

i   Inserts text before the cursor.

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 CTRL-J (linefeed) and CTRL-N.

k   Up arrow. Moves the cursor up one line. CTRL-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 that is specified by the next character a–z. Return to this position or use with an operator using ' ` ' or ' ' '.

n   Repeats the last / or ? scanning commands.

o   Opens new lines below the current line; otherwise like O.

p   Puts text after or below the cursor; otherwise like P.

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.

s   Changes the single character under the cursor to the text that 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.

t   Advances the cursor up to 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.

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 that inserted text on more than one line, the lines are saved in the numeric named buffers.

v   Unused.

w   Advances to the beginning of the next word, as defined by b.

x   Deletes the single character under the cursor. With a count deletes that many characters forward from the cursor position, but only on the current line.

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.

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 before the z gives the number of the line to place in the center of the screen instead of the default current line. To change the window size, use a count after the z and before the RETURN, as in z 5<CR>.

{ Retreats to 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).

| Places the cursor on the character in the column specified by the count.

} Advances to the beginning of the next paragraph. See { for the definition of paragraph.

~ Converts the character under the cursor to lower-case if it is upper-case, or to upper-case if it is lower-case. With a count, converts the specified number of characters at and to the right of the cursor. The cursor ends up after the last character converted.

CTRL-C (DEL) Interrupts the editor, returning it to command accepting state.

2.10. Terminal Information

`vi` works on a large number of display terminals. You can edit a terminal description file to drive new terminals. While it is advantageous to have an intelligent terminal that can locally insert and delete lines and characters from the display, `vi` functions quite well on dumb terminals over slow phone lines. `vi` allows for the low bandwidth in these situations and uses smaller window sizes and different display updating algorithms to make best use of the limited speed available.

You can also use the `vi` command set on hardcopy terminals, storage tubes and 'glass ttys' using a one-line editing window.

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.

Table 2-6   *Terminal Types*

Code	Full Name	Type
sun	Sun Workstation	Intelligent
tvi925	Televideo 925	Dumb
wy-50	Wyse 50	Dumb
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
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 your terminal type:

```
hostname% setenv TERM 2621
```

If you are using the Bourne shell, use:

```
$ TERM=2621
$ export TERM
```

If you want to arrange to have your terminal type set up automatically when you log in, 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*) is

```
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*. You can use `tset` to change the erase and kill characters, too.

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`. The editor adopts the convention that a null string in the environment is the same as not being set. This applies to TERM, TERMCAP, and EXINIT.

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. If EXINIT doesn't exist, you will get all the normal defaults.

Should you inadvertently hang up the phone while inside `vi`, or should something else go wrong, all may not be lost. Upon returning to the system, type:

```
hostname% vi -r filename
```

This will normally recover the file. If there is more than one temporary file for a specific filename, `vi` recovers the newest one. You can get an older version by recovering the file more than once. The command `vi -r` without a filename lists the files from an on-line list that were saved in the last system crash (but *not* the file just saved when the phone was hung up).

Open Mode on Hardcopy Terminals and 'Glass tty's'

If you are on a hardcopy terminal or a terminal that does not have a cursor that 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 the text is displayed. In *open* mode the editor uses a single-line window into the file, and moving backward and forward in the file displays new lines, always below the current line. Two `vi` commands that work differently in *open* mode are:

□   z and

□   CTRL-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 CTRL-R command retypes the current line. On such terminals, `vi` 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 that are deleted. It 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 that can support `vi` in the full screen mode. You can do this by entering `ex` and using an *open* command.

Editing on Slow Terminals

When you are on a slow terminal, it is important to limit the amount of output that 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 noredraw<CR>
```

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 that is redrawn each time the screen is cleared by giving window size as an argument to the commands that 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 it locates.

You can expand or contract the window size, placing the current line as you choose, with the z command, as in z 5<CR>, which changes the window to five lines. You can also use . or −. Thus the command z5 . redraws the screen with the current line in the center of a five-line window. Note that the command 5z . has an entirely different effect, placing line 5 in the center of a new window. Use

−, as in 5z− to position the cursor at line 5 in the file.

The default window sizes are 8 lines at 300 baud, 16 lines at 1200 baud, and full-screen size at 9600 baud. Any baud rate less than 1200 behaves like 300, and any over 1200 like 9600.

If the editor is redrawing or otherwise updating large portions of the display, you can interrupt this updating by typing 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 typing a CTRL-L, or you can move or search through the file again, ignoring the current state of the display.

See the section on *open* mode for another way to use the vi command set on slow terminals.

Upper-Case Only Terminals

If your terminal has only upper-case characters, you can still use vi by using the normal system convention for typing on such a terminal. Characters that 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 \(\ˆ\) \! \'. These characters are represented on the display in the same way they are typed. However, the '\' character you type will not echo until you type another key.

Vi Quick Reference

Entering and Leaving vi

% vi *name*	edit *name* at top
% vi +*n name*	... at line *n*
% vi + *name*	... at end
% vi −r	list saved files
% vi −r *name*	recover file *name*
% vi *name* ...	edit first; rest via :n
% vi −t *tag*	start at *tag*
% vi +/*pat name*	search for *pat*
% view *name*	read only mode
ZZ	exit from *vi*, saving changes
CTRL-Z	stop *vi* for later resumption

The Display

Last line	Error messages, echoing input to : / ? and !, feedback about i/o and large changes.
@ lines	On screen only, not in file.
~ lines	Lines past end of file.
CTRL-*x*	Control characters, DEL is delete.
tabs	Expand to spaces, cursor at last.

Vi Modes

Command	Normal and initial state. Others return here. ESC (escape) cancels partial command.
Insert	Entered by a i A I o O c C s S R. Arbitrary text then terminates with ESC character, or abnormally with interrupt.
Last line	Reading input for : / ? or !; terminate with ESC or CR to execute, interrupt to cancel.

Counts Before vi Commands

line/column number	z G |
scroll amount	CTRL-D CTRL-U
replicate insert	a i A I
repeat effect	most rest

Simple Commands

dw	delete a word
de	... leaving punctuation
dd	delete a line
3dd	... 3 lines
*itext*ESC	insert text *abc*
cw*new*ESC	change word to *new*
ea*s*ESC	pluralize word
xp	transpose characters

Interrupting, Cancelling

ESC	end insert or incomplete cmd
CTRL-C	interrupt (or DEL)
CTRL-L	refresh screen if scrambled

File Manipulation

:w	write back changes
:wq	write and quit
:q	quit
:q!	quit, discard changes
:e *name*	edit file *name*
:e!	reedit, discard changes
:e + *name*	edit, starting at end
:e +*n*	edit starting at line *n*
:e #	edit alternate file
CTRL-^	synonym for :e #
:r *name*	read file *name*
:w *name*	write file *name*
:w! *name*	overwrite file *name*
:sh	run shell, then return
:!*cmd*	run *cmd*, then return
:n	edit next file in arglist
:n *args*	specify new arglist
:f	show current file and line
CTRL-G	synonym for : f
:ta *tag*	to tag file entry *tag*
CTRL-]	:ta, following word is *tag*

Positioning within File

CTRL-F	forward screenfull
CTRL-B	backward screenfull
CTRL-D	scroll down half screen
CTRL-U	scroll up half screen
G	goto line (end default)
/*pat*	next line matching *pat*
?*pat*	prev line matching *pat*
n	repeat last / or ?
N	reverse last / or ?
/*pat*/+*n*	n'th line after *pat*
?*pat*?−*n*	n'th line before *pat*
]]	next section/function
[[	previous section/function
%	find matching ( ) { or }

Adjusting the Screen

CTRL-L	clear and redraw
CTRL-R	retype, eliminate @ lines
zCR	redraw, current at window top
z−	... at bottom
z.	... at center
/*pat*/z−	*pat* line at bottom
z*n*.	use *n* line window
CTRL-E	scroll window down 1 line
CTRL-Y	scroll window up 1 line

Marking and Returning

``	previous context
''	... at first non-white in line
m*x*	mark position with letter *x*
`*x*	to mark *x*
'*x*	... at first non-white in line

Line Positioning

H	home window line
L	last window line
M	middle window line
+	next line, at first non-white
−	previous line, at first non-white
CR	return, same as +
↓ or j	next line, same column
↑ or k	previous line, same column

Character Positioning

^	first non-blank
0	beginning of line
$	end of line
h or →	forward
l or ←	backwards
CTRL-H	same as ←
space	same as →
f*x*	find *x* forward
F*x*	f backward
t*x*	upto *x* forward
T*x*	back upto *x*
;	repeat last f F t or T
,	inverse of ;
|	to specified column
%	find matching ({) or }

Words, Sentences, Paragraphs

w	word forward
b	back word
e	end of word
)	to next sentence
}	to next paragraph
(	back sentence
{	back paragraph
W	blank delimited word
B	back W
E	to end of W

Commands for LISP

)	Forward s-expression
}	... but don't stop at atoms
(	Back s-expression
{	... but don't stop at atoms

Corrections During Insert

CTRL-H	erase last character
CTRL-W	erases last word
erase	your erase, same as CTRL-H
kill	your kill, erase input this line
\	escapes CTRL-H, your erase and kill
ESC	ends insertion, back to command
CTRL-C	interrupt, terminates insert
CTRL-D	backtab over *autoindent*
CTRL-^D	kill *autoindent*, save for next
0CTRL-D	... but at margin next also
CTRL-V	quote non-printing character

Insert and Replace

a	append after cursor
i	insert before
A	append at end of line
I	insert before first non-blank
o	open line below
O	open above
r*x*	replace single char with *x*
R	replace characters

Operators (double to affect lines)

d	delete
c	change
<	left shift
>	right shift
!	filter through command
=	indent for LISP
y	yank lines to buffer

Miscellaneous Operations

C	change rest of line
D	delete rest of line
s	substitute chars
S	substitute lines
J	join lines
x	delete characters
X	... before cursor
Y	yank lines

Yank and Put

p	put back lines
P	put before
"*x*p	put from buffer *x*
"*x*y	yank to buffer *x*
"*x*d	delete into buffer *x*

Undo, Redo, Retrieve

u	undo last change
U	restore current line
.	repeat last change
"*d*p	retrieve *d*'th last delete

Command Reference for the ex Line Editor

This chapter provides reference material for ex, the line-oriented text editor.[9] The screen-oriented display editor vi, described in the previous chapter, is actually a separate mode of ex. This chapter describes the line-oriented part of ex. You can also use these commands with vi. For a summary of ex commands, see the "ex Quick Reference."

3.1. Using ex

ex has a set of options, which you can use to tailor ex 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, and we assume that you are running ex on a Sun Workstation.

If there is a variable EXINIT in the environment, ex executes 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. Then ex looks for .exrc in the current directory, and reads commands contained in that file, if any. Option setting commands placed in EXINIT or .exrc are executed before each editor session.

If you are running ex on a terminal, ex determines the terminal type from the TERM variable in the environment when invoked. It there is a TERMCAP variable in the environment, and the type of the terminal described there matches the TERM variable, that description is used. Also if the TERMCAP variable contains a pathname (beginning with a /), ex seeks the description of the terminal in that file, rather than in the default /etc/termcap.)

The standard ex command format follows. Brackets '[' ']' surround optional parameters here.

```
ex [ - ] [-v] [-t tag] [-r] [-l] [-wn] [-x] [-R]
    [+command ] filename ...
```

The most common case edits a single file with no options, that is:

---

[9] The material in this chapter is derived from *Ex Reference Manual*, W.N. Joy, M. Horton, University of California, Berkeley.

```
hostname% ex filename
```

The '−' command line option suppresses all interactive-user feedback and is useful in processing ex 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.

Use the −r option to recover a file after an editor or system problem, retrieving the last saved version of the named file or, if no file is specified, displaying 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) in the *SunOS Reference Manual* for details). The −R option sets the *readonly* option at the start. If set, writes will fail unless you use an ! after the write. This option affects ZZ, *autowrite* and anything that writes to guarantee you won't clobber a file by accident. *Filename* 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, it defaults to '$', initially positioning ex at the last line of the first file. Other useful commands here are scanning patterns of the form '/*pat*' or line numbers, such as +100, which means 'start at line 100.'

3.2. File Manipulation

The following describes commands for handling files.

Current File

ex normally edits the contents of a single file, whose name is recorded in the *current* filename. ex performs all editing actions in a buffer 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 you write the buffer contents out to the file with a *write* command. 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 filename, 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 filename, 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*, ex will not normally write on it if it already exists. The *file* command will say [Not edited] if the current file is not considered edited.

Alternate File

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

Filename Expansion

You may specify filenames within the editor using the normal Shell expansion conventions. In addition, the character % in filenames is replaced by the *current* filename and the character # by the *alternate* filename. 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.

3.3. 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 the caret (^) and dollar sign ($) characters, 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.

Multiple Files and Named Buffers

If more than one file is given on the ex command line, the first file is edited as described above. The remaining arguments are placed with the first file in the *argument list*. You can display the current argument list with the *args* command. To edit the next file in the argument list, use the *next* command. You may also respecify the argument list 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.

To save 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*. 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.

Read-Only Mode

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 with :w *newfilename*, or can use the :w! form of write, even while in read only mode.

3.4. Exceptional Conditions

The following describes additional editing situations.

Errors and Interrupts

When errors occur ex flashes the workstation screen and displays an error diagnostic. If the primary input is from a file, editor processing terminates. If you interrupt ex, it displays 'Interrupt' and returns to its command level. If the primary input is a file, ex exits when this occurs.

Recovering If Something Goes Wrong

If something goes wrong and the buffer has been modified since it was last written out, or if the system crashes, either the editor or the system (after it reboots) attempts 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 problem. To recover a file, use the −r option. If you were editing the file *resume* for example, change to the directory where you were when the problem occurred, and use ex with the −r (recover) option:

```
hostname% ex −r file
```

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 the system has gone down. Use the −r option without a following filename:

```
hostname% ex −r
```

to display a list of the files that have been saved for you. In the case of a hangup, the file will not appear in the list, although it can be recovered.

3.5. Editing Modes

ex has five distinct modes. The primary mode is *command* mode. You type in commands in command mode when a ':' prompt is present, and execute them each time you send a complete line. In *insert* mode, ex gathers input lines and places them in the file. The *append*, *insert*, and *change* commands use insert mode. No prompt is displayed when you are in text input mode. To leave this mode and return to command mode, type a '.' alone at the beginning of a line.

The last three modes are *open* and *visual* modes, entered by the commands of the same names, and, within open and visual modes *text insertion* mode. In *open* and *visual* modes, you do local editing operations on the text in the file. The *open* command displays one line at a time on the screen, while *visual* works on the workstation and CRT terminals with random positioning cursors, using the screen as a single window for file editing changes. See the chapter on "Using vi, The Visual Display Editor" for descriptions of these modes.

3.6. Command Structure

Most command names are English words; you can use initial prefixes of the words as acceptable abbreviations. The ambiguity of abbreviations is resolved in favor of the more commonly used commands. As an example, the command substitute can be abbreviated as s while the shortest available abbreviation for the set command is se. See the "Command Reference" section for descriptions and acceptable abbreviations.

**Specifying 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. Counts are rounded down if
necessary. Thus the command 10p displays the tenth line in the buffer, while
d5 deletes five lines from the buffer, starting with the current line.

Some commands take other information or parameters, that you provide after the
command name. Examples would be option names in a set command such as,
set number, a filename in an edit command, a regular expression in a sub-
stitute command, or a target address for a copy command, such as, 1,5
copy 25.

Invoking Command Variants

A number of commands have two distinct variants. The variant form of the com-
mand is invoked by placing an ! immediately after the command name. You can
control some of the default variants with options; in this case, the ! serves to tog-
gle the default.

Flags After Commands

You may place the characters #, p and l after many commands. You must pre-
cede a p or l by a blank or tab except in the single special case of dp. The com-
mand that these characters abbreviates is executed after the command completes.
Since ex normally shows the new current line after each change, p is rarely
necessary. You can also give any number of + or − characters with these flags.
If they appear, the specified offset is applied to the current line value before the
display command is executed.

Writing Comments

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

**Putting Multiple Commands
on a Line**

You can place more than one ex command on a line by separating each pair of
commands by a pipe (|) character. However the *global* commands, comments,
and the shell escape ! must be the last command on a line, as they are not ter-
minated by a |.

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 you may quickly and
easily reverse them with *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.

3.7. Addressing Primitives

The following describes the editor commands called *addressing primitives*.

.   The current line. The current line is traditionally called 'dot' because you address it with a dot '.'. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, so you rarely use '.' 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. The forms .+3 +3 and +++ are all equivalent; if the current line is line 100, they all address line 103.

/pat/ ?pat?   Scan forward and backward respectively for a line containing *pat*, a regular expression (as defined below in the section "Regular Expressions and Substitute Replacement Patterns." The scans normally wrap around the end of the buffer. If all that is desired is to show the next line containing *pat*, you may omit trailing / or ?. If you omit *pat* or leave it explicitly empty, the last regular expression specified is located. The forms \/ and \? scan using the last regular expression used in a scan; after a substitute, // and ?? would scan using the substitute's regular expression.

'' 'x   Before each non-relative motion of the current line '.', the previous current line is marked with a tag, subsequently referred to as ''' ', which makes it easy to refer or return to this previous context. You can also establish marks with the mark command using single lower-case letters. For example, ''x' would refer to the line marked *x*.

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 you give more addresses than the command requires, 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. Null address specifications are permitted in a list of addresses; the default in this case is the current line '.'. So ',100' is equivalent to '.,100'. It is an error to give a prefix address to a command which expects none.

3.8. Regular Expressions and Substitutions

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. The editor 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 regular expression, that is // (forwards) or ?? (backwards).

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 backslash (\) to use them as "ordinary" characters. With *nomagic*, the default for edit, regular expressions are much simpler because there are only two metacharacters: '^' (beginning of line) and '$' (end of line). 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 the setting of this option is *magic*.[10]

Basic Regular Expression Summary

The following basic constructs are used to construct regular expressions (only when in *magic* mode).

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,' that is 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 a set of characters 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 a ^, the construct matches all but those characters; thus [^a–z] matches anything but a lower-case letter and of course a newline. You must escape any of the characters ^, [, or — in *string* with a preceding \.

---

[10] To discern what is true with *nomagic* it is sufficient 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.

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 regular expressions mentioned above that match a single character may be followed by the character * to form a regular expression that 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.

Substitute Replacement Patterns

If the replacement pattern is % by itself, it refers to the previous replacement pattern.

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 escape character \. The sequence ‘\n’ is replaced by the text matched by the *n*-th regular subexpression enclosed between \(and \).[11] 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.

3.9. Command Reference

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

> *address command ! parameters count flags*

All parts are optional; the simplest case is the empty command, which displays the next line in the file. To avoid confusion 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

.

---

[11] When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of \(starting from the left.

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].

(. , .) 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' (zero). 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. Same as `:vi` *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. A 'sensible' file 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 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. If executed from within *open* or *visual*, the current line is initially the first line of the file.

edit ! *file*   abbr: e
ex ! *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, for example: +/*pat*.

file   abbr: f

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. In the rare case that the current file is [Not edited] this is also noted. You have to use `w!` to write to the file, since `ex` does not want to `write` a file unrelated to the current contents of the buffer.

file *file*

The current filename 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 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` command 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.

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*   abbr: l

Prints the specified lines in a more unambiguous way: tabs are printed as CTRL-I (^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 the "Macros" section in the chapter "Using vi, the Visual Display Editor" for more details.

(.)mark *x*   abbr: ma

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 that 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. The *count* option specifies the number of lines to print.

(.) 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 the chapter on "Using vi the Visual Display Editor."

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 (hexadecimal 0x7f) is represented as ^?. The *count* option specifies the number of lines to print. 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. 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. 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*. ex also warns you if there are more files in the argument list. Normally, you do want to save your changes, so you should use 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' (zero) 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. Within *open* and *visual* modes the current line is set to the first line read rather than the last.

(.) read   ! *command*

Reads the output of *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 an accidental hangup of the phone or a system crash or preserve command. The system saves a copy of the file you were editing only if you have made changes to the file. Except when you use preserve you will be notified by mail when a file is saved.

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*   abbr: se

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 from 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.

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 */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 a y one can cause the substitution to be performed, any other input causes no change to take place. After a substitute command is executed, the last line substituted becomes the current line.

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.

(.,.)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.

tag *tag*   abbr: ta

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. When a tag command is specified with no tag, the previous tag is reused.

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* were 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 commands.) 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 its 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 that 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.

vi *file*

Same as :edit *file* or :ex *file*.

(.)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 chapter "Using vi, the Visual Display Editor" for more details. To exit visual 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.[12] If the file does not exist it

---

[12] 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 /dev/tty or /dev/null. Otherwise, you must give the variant form w! to force a write.

is created. The current file name is changed only if there is no current file name; the current line is never 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*   abbr: x

If any changes have been made and not written, writes the buffer out. Then, in any case, quits. Same as wq, but does not bother to write if there have not been any changes to the file.

(.,.)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*

Displays a window of text with the specified line at the top. If *type* is '−' the line is placed at the bottom; a '.' places the line 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 terminal, the screen is cleared before display begins unless you give a *count* less than the screen size. The current line is left at the last line displayed. 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.

! *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.

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 > shifts 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.

CTRL-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.

(. , .)
(. , .) |

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.

3.10. Option Descriptions

Here are the various options that can be set by means of the `set` command. Note that binary options can be turned off with two letters `no` in front of the option.

autoindent, ai   default: noai

The *autoindent* option 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 you then type in lines of text, 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 be aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop, type CTRL-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 CTRL-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 a ^ and immediately followed by a CTRL-D. This causes the input to be repositioned at the beginning of the line, but retains the previous indent for the next line. Similarly, a '0' (zero) followed by a CTRL-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 CTRL-^ (switch files) or CTRL-] (tag goto) command in *visual* mode. 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 `rewind`, `stop!` for `stop`, `tag!` for `tag`, `shell` for `!`, and `:e #` and a `:ta!` command from within *visual* mode).

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.

directory, dir   default: dir=/tmp

Specifies the directory in which ex places its buffer file. If this directory is not writeable, then the editor will exit abruptly when it fails to be able to create its buffer there. This feature is useful on systems where /tmp fills up. Being able to specify that the editor use your own file space can allow you to edit even if /tmp is full.

edcompatible   default: noedcompatible

Causes the presence or absence of g and c suffixes on substitute commands to be remembered, and to be toggled by repeating the suffixes. The suffix r makes the substitution be as in the ~ command, instead of like &.

errorbells, eb   default: noeb

Error messages are preceded by a beep or bell.[13] 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.

flash, fl default: fl

If the terminal has visual bell capability, use that instead of the audible 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* modes are modified to have meaning for LISP.

list   default: nolist

All printed lines will be displayed (more) unambiguously, showing tabs and ends-of-lines as in the list command.

magic   default: magic for ex and vi[14]

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 backslash (\).

---

[13] Beeping and bell ringing in *open* and *visual* on errors is not suppressed by setting *noeb*.

[14] *nomagic* for edit.

sun microsystems

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.

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 LIpplpipnpbp

Specifies the paragraphs for the { and } operations in *open* and *visual* modes. 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 :.

readonly, ro   default: off

If set, writes will fail unless you use an ! after the write. Affects x, ZZ, *autowrite* and anything that writes to guarantee you won't clobber a file by accident. Abbreviate to **ro**.

redraw   default: noredraw

The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal (e.g. during insertions in *visual* mode 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. Can map q to # and #1 to something else, and q1 to something else. If off, can map CTRL-L to 1 and CTRL-R to CTRL-L without having CTRL-R map to 1.

report   default: report=5[15]

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

---

[15] 2 for `edit`.

this same threshold. Thus notification is suppressed during a `global` command on the individual commands performed.

scroll   default: scroll=½ 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=NHSHH HUuhsh+c

Specifies the section macros for the `[ [` and `] ]` operations in *open* and *visual* modes. The pairs of characters in the option's string are the names of the macros that 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.

shiftwidth, sw   default: sw=8

Gives the width a software tab stop, used in reverse tabbing with CTRL-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.

showmode, smd default: nosmd

In *visual* mode, show the current input mode on the message line.

state	display
command mode	(nothing)
in a command	APPEND MODE
in s command	SUBSTITUTE MODE
in c command	CHANGE MODE
in R command	REPLACE MODE
in o command	OPEN MODE
in i command	INSERT MODE
in r command	REPLACE 1 CHAR

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 the chapter "Using `vi`, the Visual Display Editor" 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, similar to the *path* variable of *csh*. Separate the files by spaces, and precede each space with a backslash. Files are searched left to right. Always put *tags* as your first entry. 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   default: from environment TERM

The terminal type of the output device.

terse   default: noterse

Shorter error diagnostics are produced for the experienced user.

timeout   default: on

Causes macros to time out after one second. Turn it off and they wait forever. Use this if you want multi-character macros. If your terminal sends an escape sequence for arrow keys, type ESC twice.

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. Can specify a 12-line window at 300 baud and a 23-line window at 1200 in your EXINIT with: :set w300=12 w1200=23. Synonymous with *window* but only at 300, 1200, and 9600 baud.

wrapscan, ws   default: ws

Searches using the regular expressions in addressing will wrap around past the end of the file.

wrapmargin, wm   default: wm=0

Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. Any number other than 0 (zero) is a distance from the right edge of the area where wraps can take place. If you type past the margin, the entire word is rewritten on the next line. Behaves much like fill/nojustify mode in `nroff`. See the section "Using `vi`, the Visual Display Editor" 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.

3.11. 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* modes, 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 250,000 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.

Ex Quick Reference

Entering/Leaving ex

% ex *name*	edit *name*, start at end
% ex +*n name*	... at line *n*
% ex −t *tag*	start at *tag*
% ex −r	list saved files
% ex −r *name*	recover file *name*
% ex *name* ...	edit first; rest via :n
% ex −R *name*	read only mode
: x	exit, saving changes
: q!	exit, discarding changes

ex States

Command	Normal and initial state. Input prompted for by :. Your kill character cancels partial command.
Insert	Entered by a i and c. Arbitrary text then terminates with line having only . character on it or abnormally with interrupt.
Open/visual	Entered by open or vi, terminates with Q or ↖

ex Commands

abbrev	ab	next	n	unabbrev	una
append	a	number	nu	undo	u
args	ar	open	o	unmap	unm
change	c	preserve	pre	version	ve
copy	co	print	p	visual	vi
delete	d	put	pu	write	w
edit	e	quit	q	xit	x
file	f	read	re	yank	ya
global	g	recover	rec	window	z
insert	i	rewind	rew	escape	!
join	j	set	se	shift	<
list	l	shell	sh	print next	CR
map	map	source	so	resubst	&
mark	ma	stop	st	rshift	>
move	m	substitute	s	scroll	^D

ex Command Addresses

n	line *n*	/*pat*	next with *pat*
.	current	?*pat*	previous with *pat*
$	last	*x-n*	*n* before *x*
+	next	*x,y*	*x* through *y*
−	previous	´*x*	marked with *x*
+*n*	*n* forward	´´	previous context
%	1,$		

Specifying Terminal Type

% setenv TERM *type*	(for *csh*)
$ TERM=*type*; export TERM	(for *sh*)
See also *tset* in the user's manual.	

Some Terminal Types

2621	43	adm31	dw1	h19
2645	733	adm3a	dw2	i100
300s	745	c100	gt40	mime
33	act4	dm1520	gt42	owl
37	act5	dm2500	h1500	t1061
4014	adm3	dm3025	h1510	vt52

Initializing Options

EXINIT	place set's here in environment var.
set *x*	enable option
set no*x*	disable option
set *x=val*	give value *val*
set	show changed options
set all	show all options
set *x*?	show value of option *x*

Useful Options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
lisp		( ) { } are s-exp's
list		print ^I for tab, $ at end
magic		. [* special in patterns
number	nu	number lines
paragraphs	para	macro names which start ...
redraw		simulate smart terminal
scroll		command mode lines
sections	sect	macro names ...
shiftwidth	sw	for < >, and input ^D
showmatch	sm	to) and } as typed
slowopen	slow	choke updates during insert
window		visual mode lines
wrapscan	ws	around end of buffer
wrapmargin	wm	automatic line splitting

Scanning Pattern Formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str*]	any char in *str*
[↑*str*]	... not in *str*
[*x−y*]	... between *x* and *y*
*	any number of preceding

Using the ed Line Editor

This chapter describes the editing tools of the ed line editor.[16] It provides the newcomer with elementary instructions and exercises for learning the most necessary and common commands and the more advanced user with information about additional editing facilities. The contents include descriptions of appending, changing, deleting, moving, copying and inserting lines of text; reading and writing files; displaying your files; context searching; the global commands; line addressing; and using special characters. There are also brief discussions on writing scripts and on the pattern-matching tool grep, which is related to ed.

We assume that you know how to log in to the system and that you have an understanding of what a file is. You must also know what character to type as the end-of-line on your workstation or terminal. This character is the RETURN key in most cases.

If you need basic information on the Sun system, refer to *SunOS User's Guide: Getting Started*. See ed(1) in the *SunOS Reference Manual* for a nutshell description of the ed commands.

4.1. Getting Started

The ed text editor is an interactive program for creating and modifying text, using directions that you provide from your workstation. The text can be a document, a program or perhaps data for a program.

We'll assume that you have logged in to your system, and it is displaying the hostname and prompt character, which we show throughout this manual as:

```
hostname%
```

To use ed, type ed and a carriage return at the 'hostname%' prompt:

```
hostname% ed
```

You are now ready to go. ed does not prompt you for information, but waits for you to tell it what to do. First you'll learn how to get some text into a file and later how to change it and make corrections.

---

[16] The material in this chapter is derived from *A Tutorial Introduction to the UNIX Text Editor*, B.W. Kernighan and *Advanced Editing on UNIX*, B.W. Kernighan, Bell Laboratories, Murray Hill, New Jersey.

Creating Text — the Append Command a

Let's assume you are typing the first draft of a memo and starting from scratch. When you first start ed, in this case, you are working with a 'blank piece of paper'; there is no text or information present. To supply this text, you either type it in or read it in from a file. To type it in, use the *append* command a.

So, to type in lines of text into the buffer, you type an a followed by a RETURN, followed by the lines of text you want, like this:

```
hostname% ed
a<CR>
Now is the time
for all good men
to come to the aid of their party.
.
```

If you make a mistake, use the DEL key to back up over and correct your mistakes. You cannot go back to a previous line after typing RETURN to correct your errors. The only way to stop appending is to tell ed that you have finished by typing a line that contains only a period. It takes practice to remember it, but it has to be there. 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; you will have to take them out later.

After the append command, your file contains the 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, type another a, and continue typing.

If you have not used a text editor before, read the following to learn a bit of terminology. If you have used an editor, skip to the "Error Messages — ?" section.

In ed jargon, the text being worked on is said to be in a work space or 'kept in a buffer'. The buffer is like a piece of paper on which you write things, change some of them, and finally file the whole thing away for another day.

You have learned how to tell ed what to do to the text by typing instructions called *commands*. Most commands consist of a single letter that you type in lower case letters. An example is the append command a. Type each command on a separate line. You sometimes precede the command by information about what line or lines of text are to be affected; we discuss this shortly.

As you have seen, ed does not respond to most commands; that is, there isn't any prompting or message display like 'ready'. If this bothers you as a beginner, be patient. You'll get used to it.

Error Messages — ?

When you make an error in the commands you type, ed asks you:

```
?
```

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

When you want to save your text for later use, write out the contents of the buffer into a file with the *write* command w, followed by the filename you want to write in. The w command copies the buffer's contents into the specified file, destroying any previous information on the file. To save the text in a file named *junk*, for example, type:

```
w junk
68
```

Leave a space between the w and the filename. ed responds by displaying the number of characters it wrote out, in this case 68. Remember that blanks and the return character at the end of each line are included in the character count. The buffer's contents are not disturbed, so you can go on adding lines to it. This is an important point. ed works on a copy of a file at all times, not on the file itself. There is no change in the contents of a file until you type a w. Writing out the text into a file from time to time is a good idea to save most of your text should you make some horrible mistake. If you do something disastrous, you only lose the text in the buffer, not the text that was written into the file.

When you want to copy a portion of a file to another name so you can format it separately, use the w command. Suppose that in the file being edited you have:

```
.TS
       ...lots of stuff
.TE
```

This is the way a table is set up for the tbl program. To isolate the table in a separate file called, for example, *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 w can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; give one line number instead of two (we explain line numbers later — see the section "Specifying Lines in the

Editor" for details). For example, if you have just typed a very long, complicated line and you know that you are going to need it or something like it later, then save it — don't re-type it. In the editor, say:

```
a
...lots of stuff...
...very long, complicated line...
.
.w   temp
number of characters
a
...more stuff...
.
.r temp
number of characters
a
...more stuff...
.
```

This last example is worth studying to be sure you appreciate what's going on. The `.w temp` writes the very long, complicated line (the current line) you typed to the file called *temp*. The `.r temp` reads that line from *temp* into the file you are editing after the current line 'dot' so you don't have to re-type it.

Leaving ed — the Quit Command q

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

```
w
number of characters
q
hostname%
```

The system responds with the hostname prompt. At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting. Actually, `ed` displays '?' if you try to quit without writing. At that point, write the file if you want; if not, type another `q` to get you out of `ed` regardless of whether you changed the file or not.

Creating a New File — the Edit Command e

The `edit` command `e` says "I want to edit a new file called *newfile*, without leaving the editor." To do this, you type:

```
e   newfile
```

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 filename, except that if you have a pattern remembered, a command like `//` will still work. (See the section "Repeated Searches — `//` and `??`" later in this chapter.)

If you enter ed with the command:

```
hostname% ed file
```

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

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

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with w in a previous session. The edit command e also fetches the entire contents of a file into the buffer. So if you had saved the three lines 'Now is the time', etc., with w in an earlier session, the ed command e fetches the entire contents of the file *junk* into the buffer, and responds with the number of characters in *junk*:

```
hostname% e junk
68
```

If anything was already in the buffer, it is deleted first.

If you use e to read a file into the buffer, you do not need to use a filename after a subsequent w command; ed remembers the last filename used in an e command, and w will write on this file. Thus a good way to operate is:

```
hostname% ed
e file
number of characters
[editing session]
w
number of characters
q
hostname%
```

This way, you can simply say w from time to time, and be secure that you are writing into the proper file each time.

Exercise: Trying the e Command

Experiment with the e command — try reading and displaying various files. You may get an error

```
?name
```

where *name* is the name of a file; this means that the file doesn't exist, typically because you spelled the filename wrong, or perhaps because you are not allowed to read or write it. Try alternately reading and appending to see that they work similarly. Verify that:

```
hostname% ed filename
number of characters in file
```

is equivalent to:

```
hostname% ed
e filename
number of characters in file
```

Checking the Filename — the Filename Command f

You can find out the remembered filename at any time with the f command; just type f without a filename. In this example, if you type f, ed replies:

```
hostname% ed junk
68
f
junk
```

You can also change the name of the remembered filename with f; this following sequence guarantees that a careless w command will write on *junk* instead of *precious*. Try:

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

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. To do this, use the *read* command r. The command:

```
r junk
68
```

reads the file *junk* into the buffer, adding it to the end of whatever is already in the buffer. ed responds with the number of characters in the buffer. So if you do a read after an edit:

```
hostname% ed junk
68
r junk
68
w
136
q
hostname%
```

the buffer contains *two* copies of the text or six lines (136 characters) in this case. Like w and e, r displays the number of characters read in after the reading operation is complete. Now check the file contents with cat:

```
hostname% cat junk
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.
hostname%
```

Generally speaking, you won't use r as much as e.

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 ...

The data contained in *table* has to go there so nroff or troff will format it properly. Now what?

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

```
hostname% 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, which is the same as $r.

Printing the Buffer Contents — the Print Command p

To print or 'display' the contents of the buffer or parts of it on the screen, use the *print* command p. To do this, specify the lines where you want the display to begin and where you want it to end, separated by a comma, and followed by p. Thus to show the first two lines of the buffer, for example, say:

```
1,2p     (starting line=1, ending line=2 p)
Now is the time
for all good men
```

Suppose you want to print *all* the lines in the buffer. You could use 1, 3p if you knew there were exactly three lines in the buffer. But in general, you don't know how many lines 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 to display *all* the lines in the buffer, line 1 to last line:

```
1,$p
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.
```

If you want to stop the display of more than one screenful before it is finished, type the interrupt character — probably (Control-C).

```
CTRL-C
?
```

ed waits for the next command.

To display the *last* line of the buffer, you can use:

```
$,$p
to come to the aid of their party.
```

or abbreviate it to:

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

You can show any single line by typing the line number followed by a p. So, to display the first line of the buffer, type:

```
1p
Now is the time
```

In fact, ed lets you abbreviate even further: you can display any single line by typing *just* the line number — there is no need to type the letter p. So if you say:

```
2
for all good men
```

ed displays the second line of the buffer.

You can also use $ in combinations to display the last two lines of the buffer, for example:

```
$-1,$p
for all good men
to come to the aid of their party.
```

This helps when you want to see how far you got in typing.

Exercise: Trying the p Command

As before, create some text using the a command and experiment with the p command. You will find, for example, that you can't show line 0 or a line beyond the end of the buffer, and that attempts to show a buffer in reverse order don't work. For example, you get an error message if you type:

```
3,1p
?
```

Displaying Text — the List Command l

ed provides two commands for displaying the contents of the lines you're editing. You are familiar with the p command that displays lines of text. Less familiar is the *list* command l (the letter 'ell'), 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 show each tab as > and each backspace as <. A sample display of a random file with tab characters and backspaces is:

```
l
Now is the >> time for << all good men
```

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 displayed 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 displaying long lines on small terminal screens. A sample output of a folded line is:

```
l
This is an example of using the l command to display a very long line \
that has more than 72 characters ...
```

Occasionally the l command displays in a line a string of numbers preceded by a backslash, such as '\07' or '\16'. These combinations make visible the characters that normally don't show, 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 displayed on some terminals. Often their

presence means that your finger slipped while you were typing; you almost never want them.

The Current Line — 'Dot' or '.'

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

```
1,3p
Now is the time
for all good men
to come to the aid of their party.
```

ed has displayed the three lines for you. Try typing just a p to display:

```
p      (no line numbers)
to come to the aid of their party.
```

The line displayed is the third line of the buffer. In fact it is the last or most recent line that you have done anything with. (You just displayed it!) You can repeat p without line numbers, and it will continue to display 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 displayed) so that you can use it instead of an explicit line number. You refer to this most recent line by the shorthand symbol:

```
.      (pronounced 'dot')
to come to the aid of their party.
```

Dot is a line number in the same way that '$' is; it means exactly 'the current line', or loosely, 'the line you most recently did something to'. You can use it in several ways — one possibility is to display all the lines from and including the current line to the end of the buffer.

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

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 displayed; that is, after this command sets both '.' and '$' refer to the last line of the file, line 6.

Dot is most useful in combinations like:

```
.+1 (or equivalently, .+1p)
```

This means 'show the next line' and is a handy way to step slowly through a

buffer. You can also say:

```
.-1     (or .-1p)
```

This means 'show the line *before* the current line'. Use this to go backward if you wish. Another useful one is something like:

```
.-3,.-1p
```

This command displays 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:

```
.=
3
```

Let's summarize some things about p and dot. Essentially you can precede p by 0, 1, or 2 line numbers. If you do not give a line number, p shows the 'current line', the line that dot refers to. If there is one line number given with or without the letter p, it shows that line and dot is set there; and if there are two line numbers, it shows all the lines in that range, and sets dot to the last line displayed. If you specify two line numbers, the first can't be bigger than the second.

Typing a single RETURN displays 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 Delete Command d

Suppose you want to get rid of the three extra lines in the buffer. To do this, use the delete command d. The d command is similar to p, except that d deletes lines instead of displaying them, You specify the lines to be deleted for d exactly as you do for p:

```
starting line, ending line d
```

Thus the command:

```
4,$d
```

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

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

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: Experimenting

Experiment with a, e, r, w, p and d until you are sure you know what they do, and until you understand how to use dot, '$' and the line numbers.

If you are adventurous, try using line numbers with a, r and w as well. You will find that a appends lines *after* the line number that you specify rather than after dot; that r reads a file in *after* the line number you specify and not necessarily at the end of the buffer; and that w writes out exactly the lines you specify, not necessarily the whole buffer. These variations are useful, for instance, for inserting a file at the beginning of a buffer:

```
0r filename
number of characters
```

ed indicates the number of characters read in. You can enter lines at the beginning of the buffer by saying:

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

Or you can write out the lines you specify with w. Notice that .w is *very* different from:

```
.
w
number of characters
```

Modifying Text — the Substitute Command s

One of the most important commands is the *substitute* command s. Use s to change individual words or letters within a line or group of lines. For example, you can correct 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'. ed does not display the result automatically, so verify that it works with:

```
p
Now is the time
```

You get what you wanted. Notice that dot has been set to the line where the substitution took place, since p printed that line. The s command always sets dot in this way.

The general way to use the substitute command is:

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

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

Thus you can say:

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

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

You can precede any s command by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus, to change the *first* occurrence of 'mispell' to 'misspell' on every line of the file, type:

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

But to change *every* occurrence in every line, type:

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

This is more likely what you wanted in this particular case.

Note: Be careful that this is exactly what you want to do. Unless you specify the substitution specifically, globally changing the string 'the', will also change every instance of those characters, including 'o*the*r', etc.

If you do not give any line numbers, s assumes you mean 'make the substitution on line dot,' so it changes things only on the current line. You will see that a very common sequence is to correct a mistake on the current line, and then display the line to make sure everything is all right:

```
s/something/something else/p
line with something else
```

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.*

You can also say:

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

which means 'change the first string of characters to *nothing*;' that is, remove the first string of characters. Use this sequence for deleting extra words in a line or removing extra letters from words. For instance, if you had:

```
Nowxx is the time
```

To correct this, say:

```
s/xx//p
Now is the time
```

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

If you want to replace the *first* 'this' on a line with 'that', for example, use:

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

If there is more than one 'this' on the line, a second form with the trailing *global* command g changes *all* of them:

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

The general format is:

```
s/ ... / ... /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. If you get funny results using any of the characters:

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

read the section on "Special Characters."

You can follow either form of the s command by p or l to display or list the contents of the line.

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

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

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 was changed will be displayed, not all the lines. We will talk later about how to show all the lines that were modified.

The Ampersand &

The & is a shorthand character — it is used only on the right-hand part of a substitute command where it means 'whatever was matched on the left-hand side'. Use it to save typing. Suppose the current line contained:

```
Now is the time
```

and you wanted to put parentheses around it. You could just retype the line, but this is tedious. Or you could say:

```
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'.

You can use the & several times in a line:

```
s/.*/&?  &!!/
Now is the time?  Now is the time!!
```

or

```
s/the/& best and & worst/
Now is the best and the worst 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/
the end of the world is at hand
```

Observe this expression carefully, for 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.

Notice that & is not special on the left side of a substitute, only on the *right* side.

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 backslash (\):

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

converts the word 'ampersand' into the literal symbol '&' in the current line. 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 put parentheses around a line, regardless of its length, use:

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

Exercise: Trying the s and g Commands

Experiment with s and g. 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
on the other side of the coin
```

Undoing a Command — the Undo Command u

Occasionally you will make a substitution in a line, only to realize too late that it was a mistake. Use the *undo* command u to undo the last substitution. This restores the last line that was substituted to its previous state. For example, study the following example:

```
s/party/country/
p
to come to the aid of their country.
u
p
to come to the aid of their party.
```

4.2. Changing and Inserting Text — the c and i Commands

This section discusses the *change* command c and the *insert* command i. The change command changes or replaces a group of one or more lines. The insert command inserts a group of one or more lines.

The c command replaces a number of lines with different lines you type in at the workstation. 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 '.' take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines that have errors in them.

If you only specify one line in the c command, just that line is replaced. You can type in as many replacement lines as you like. Notice the use of '.' to end 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, for instance:

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

inserts the given text *before* the next line that contains 'string', that is, 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: Trying the c Command

Change is rather 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 to append *after* the given line, you type:

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

.
```

while to insert *before* it, you type:

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

Observe that if you do not give a line number, i inserts before line dot, while a appends after line dot.

4.3. Specifying Lines in the Editor

To specify which lines are to be affected by the editing commands, you use *line addressing*. There are several methods, and they are described below.

Context Searching

One way is *context searching*. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

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.
```

If you want to find the line that contains 'their' so you can change it to 'the'. 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 contains several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be.

For example, to locate the next occurrence of the characters between slashes ('their'), type:

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

To search for a line that contains a particular string of characters, the general format is:

```
/ string of characters we want to find /
```

This is sufficient to find the desired line. It also sets dot to that line and displays the line for verification. '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 displays the error message:

```
?
```

Otherwise it shows the line it found.

Less familiar is the use of:

```
?thing?
```

This command scans *backward* 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. You can do both the search for the desired line *and* a substitution all at once, like this:

```
/their/s/their/the/p
to come to the aid of the party.
```

There were three parts to that last command: a context search for the desired line, the substitution, and displaying the line.

The expression /their/ is a context search expression. In their simplest form, all context search expressions are like this — a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so you can use them by themselves to find and show a desired line, or as line numbers for some other command, like s. We use them both ways in the examples above.

Exercise: Trying Context Searching

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. You can also use context searching with r, w, and a.

If you get funny results with any of the characters:

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

read the section on "Special Characters."

Specifying Lines with Address Arithmetic — + and −

Another area where you can save typing in specifying lines is to use minus (−) and plus (+) as line numbers by themselves. To move back up one line in the file, type:

```
−
```

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, use it to change 'bad' to 'good' on the previous line and on the current line.

```
−,.s/bad/good/
```

You can use + and − in combination with searches using /.../ and ?...?, and with $. To find the line containing 'thing', and position you two lines before it, type:

```
/thing/−−
```

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

```
$−1
```

displays 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, type:

```
$−5,$p
```

which shows the last six lines. (Be sure you understand why it shows six, not five.) If there are less than six, of course, you'll get an error message. 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/
```

Convenience dictates the choice. You could display all three lines by, for instance:

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

or:

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

or by any number of similar combinations. The first one of these might be better if you don't know how many lines are involved. Of course, if there were only three lines in the buffer, you'd use:

```
1,$p
```

but not if there were several hundred.

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

As another example:

```
.-3,.+3p
```

displays 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, you can omit the '+':

```
.-3,.3p
```

is identical in meaning.

Repeated Searches — // and ??

Suppose you ask for the search:

```
/horrible thing/
```

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

```
//
```

as a shorthand for 'the previous thing that was searched for', whatever it was. You can repeat this as many times as necessary. You can also search backward through the file by typing:

```
??
```

?? 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 backward and change a line, say:

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

You can still use the & on the right hand side of a substitute to stand for whatever got matched:

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

This finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then displays 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, that is, 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 give a search command like:

```
/thing/
```

you are left pointing at the next line that contains 'thing'. No address is required with commands like s to make a substitution on that line. Addresses are also not required with p to show it, 1 to list it, d to delete it, a to append text after it, c to change it, or i to insert text before it.

What would happen if there were no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you are sitting on the only 'thing' when you issue 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 do not give a line number with them, the a appends text after the current line, c changes the current line, and i inserts text before the current line.

The a, c, and i commands 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)
.                        (to get out of append mode)
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)
.                               (to get out of append mode)
c                               (replace entire line)
... fixed up line ...
```

You should experiment to determine what happens if you do not add *any* lines

with a, c or i.

The r command reads a file into the text being edited, either at the end if you do not give an 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 that involves a context search, such as:

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

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

With the s command, the rule is simple; you are left positioned on the last line that got changed. If there were no changes, dot doesn't move.

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

```
x1
x2
x3
```

The command line

```
-,+s/x/y/p
```

displays the third line, 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 show only the first line, and that is where dot would be set.

Combining Commands — the 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
```

would display 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 display 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 you cannot display lines in reverse order.

This happens 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, you can use the semicolon ; 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 the example above, the command:

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

displays 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.

Use the semicolon when you want to find the *second* occurrence of something. For example, to find the second occurrence of 'thing', you can say:

```
/thing/
line with 'thing'
//
second line with 'thing'
```

But this displays 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 find the first occurrence of 'thing', set dot to that line, then find the second and display only that:

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

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

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

We leave you to try showing the third or fourth or ... in either direction.

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/
```

This search fails if 'thing' occurs on line 1. But it is possible to say:

```
0;/thing/
```

This is 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, be aware that if you type an INTERRUPT (CTRL-C is the default, but your terminal may be set up with the DELETE, RUBOUT or BREAK keys) 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 the middle of execution in some clean but unpredictable state; hence it is not usually wise to stop them. Dot may or may not be changed.

Displaying is more clear cut. Dot is not changed until the display is done. Thus if you display lines until you see an interesting one, then type ⌐Control-C⌐, you are *not* sitting on that line or even near it. Dot is left where it was when the p command was started.

4.4. Editing All Lines — the Global Commands g and v

Use the *global* command g to execute one or more ed commands on all those lines in the buffer that match some specified string. For example, to display all lines that contain 'peling', type:

```
g/peling/p
```

As another example:

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

displays all the formatting commands in a file. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; the same rules and limitations apply.

For a more useful command, which makes the substitution everywhere on the line, then displays each corrected line, type:

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

Compare this to the following command line, which only displays the last line substituted:

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

Another subtle difference is that the g command does not give a '?' if 'peling' is not found whereas the s command will.

The substitute command is probably the most useful command that can follow a global because you can use this to make a change and display each affected line for verification. For example, you can change the word 'SUN' to 'Sun' everywhere in a file, and verify that it really worked, with:

```
g/SUN/s//Sun/gp
```

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

The v command is identical to g, except that it operates on those lines that do *not* contain an occurrence of the pattern; that is, v 'inverts' the process, so:

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

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.

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/+
```

displays 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 backward until it finds a line that begins .SH (a section heading) and shows the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally:

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

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

You can also precede the g and v commands by line numbers, in which case the lines searched are only those in the range specified.

Multi-line Global Commands

You can use 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 g that the set of commands continues on the next line; it terminates on the first line that does not end with '\'. You can't use a substitute command to insert a newline within a g command.

Watch out for the command:

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

which 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 an \ 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
```

You do not need a final line containing a '.' to terminate the i command, unless you are using further commands under the global command. On the other hand, it does no harm to put it in either.

4.5. Special Characters

Certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. You may have noticed that things just don't work right when you use some characters like '.', *, $, and others in context searches and with the substitute command. These special characters are called *metacharacters*. 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 is:

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

Matching Anything — the Dot '.'

Use the 'dot' metacharacter '.' to match any single character. For example, to find any line where 'x' and 'y' occur separated by a single character, type:

```
/x.y/
```

You may get any of:

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

and so on.

Since '.' matches a single character, it gives you a way to deal with funny characters that l displays. Suppose you have a line that, when displayed with the l command, appears as:

```
    ... th07is ...
```

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 comma (,), 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'. 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:

```
.s/././
.ow is the time.
```

which is probably not what you intended.

**Specifying Any Character —
the Backslash ' \\'**

The backslash character ' \\' is special to ed as noted in the description of the ampersand. For safety's sake, avoid the backslash where possible. 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/
```

changes '\.*' into 'backslash dot star'.

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line:

```
Now is the time.
```

into:

```
Now is the time?
```

Use the backslash '\' here as well to turn 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.', type:

```
s/\./?/p
Now is the time?
```

ed treats the pair of characters '\.' as a single real period.

You can also use the backslash when searching for lines that contain a special character. Suppose you are looking for a line that contains:

```
.PP
```

The search for .PP finds:

```
/.PP/
THE APPLICATION OF ...
```

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

```
/\.PP/
```

you will find only lines that contain .PP.

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 that each 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/
```

Here are 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
```

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

Specifying the End of Line — the Dollar Sign $

The dollar-sign, $, denotes the end of a line:

```
/string$/
```

only finds an occurrence of 'string' that is at the end of some line. This implies, of course, that:

```
/^string$/
```

finds a line that contains just 'string', and:

```
/^.$/
```

finds a line containing exactly one character.

As an 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/p
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 a line with a period without altering the first comma. Type:

```
s/,$/./p
Now is the time, for all good men,
```

The $ sign here specifies the comma at the end of the sentence. Without it, of course, s operates on the first comma to produce:

```
s/,/./p
Now is the time. for all good men,
```

As another example, to convert:

```
Now is the time.
```

into:

```
Now is the time?
```

as you did earlier, you can use:

```
s/.$/?/p
Now is the time?
```

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.

Specifying the Beginning of the Line — the Circumflex ^

The circumflex ^ signifies the beginning of a line. Thus:

```
/^string/
string
```

finds 'string' only if it is at the beginning of a line, but not:

```
the string...
```

You can also use ^ to insert something at the beginning of a line. For example, to place a space at the beginning of the current line, type:

```
s/^/ /
```

You can combine metacharacters. To search for a line that contains *only* the characters .PP by typing:

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

Matching Anything — 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'.

You can use the star with any character, not just the space. If the original example was instead:

```
text  x--------y  text
```

then you can replace all – signs 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? What will happen if you blindly type:

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

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/
```

takes everything from the *first* 'x' to the *last* 'y'. In this example, this is 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'.

The dot is useful in conjunction with *, a repetition character; a* is a shorthand for 'any number of 'a' s', so .* matches any number of anythings. Use this like:

```
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.

There are times when the pattern   .* is exactly what you want. For example, use:

```
Now is the time for all good men ....
s/ for.*/./p
Now is the time.
```

The .* replaces all of the characters from the space before the word 'for' with a dot. The string 'Now is the time.' is the result in this example.

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

```
text  xy  text  x              y  text
```

and you 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:

```
/xb/b/*y/
```

where ƀ represents a blank. This describes '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 following command does not produce what was intended:

```
abcdef
s/x*/y/g
p
yaybycydyeyfy
```

The reason for this behavior again, 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'.

Character Classes — Brackets
[   ]

The [and] brackets form 'character classes'. 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. For example, to match any single digit, use:

```
/[0123456789]/
```

Any one of the characters inside the braces will cause a match. It is 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.

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 all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [and].

Another example: To match zero or more digits (an entire number), and to delete all digits from the beginning of all lines, type:

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

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.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. To do this, begin the class with a caret (^) to stand for 'any character *except* a digit':

```
[^0-9]
```

Thus you might find the first line that does not 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 to find a line that doesn't begin with a circumflex, you type:

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

4.6. Cutting and Pasting with the Editor
Moving Lines Around

ed has commands for manipulating individual lines or groups of lines in files.

There are several ways to move text around in a file.

Moving Text Around — the Move Command m

Use the *move* command m for cutting and pasting — you can move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You could do it by saying:

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

This is the brute force way; that is, you write the paragraph into a temporary file, read in the temporary file at the end, and then delete it from its current position. As another example, consider:

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

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

But you can do it a lot easier with m, so you can do a whole operation at one crack.

```
1,3m$
```

The general case is:

```
start line, end line m after this line
```

Notice that there is a third line to be specified — the place where the moved stuff gets put.

If you try:

```
1,5m3
?
```

ed reminds you that you can't do this.

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 moved. 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, dollar signs, or other ways to specify lines.

Of course you can specify the lines to be moved by context searches; if you had:

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

you could reverse the two paragraphs like this:

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

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 after the second. Suppose that you are positioned at the first. Then, to move line dot to one line after line dot, type:

```
m+
```

If you are positioned on the second line, and want to do the reverse, type:

```
m--
```

As you can see, m is more succinct and direct than writing, deleting and re-reading. When is brute force better? This is a matter of personal taste — do what you have most confidence in. The main difficulty with m 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 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 use a w command before doing anything complicated; then if you goof, it's easy to back up to where you were.

Substituting Newlines

You can split 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 nroff formatting command .ul.

```
text a very big text
```

To convert 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, type:

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

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

Joining Lines — the Join Command j

You may also join lines together, but use the *join* command j for this 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 displays it.

Rearranging a Line with \ (. . . \ )

Skip this section if this is the first time you're reading this chapter. Recall 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 provide a way for you to display exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

Marking a Line — the Mark Command k

You can mark a line with a particular name so you can refer to it later 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. To mark the current line with the name x, use:

```
kx
```

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 — the Transfer Command t

We mentioned earlier the idea of saving a line that was hard to type or used often, to cut down on typing time. Of course this can be more than one line, in which case the saving is presumably even greater.

ed provides another command, called t (*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 m, except that instead of moving lines, it simply duplicates them at the place you named. Thus, to duplicate the entire contents that you are editing, use:

```
1,$t$
```

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.

4.7. Escaping to the Shell with !

Sometimes it is convenient to be able to temporarily escape from the editor to use some Shell command without leaving the editor. Use the ! (escape) command to do this.

To suspend your current editing state and execute the shell command you asked for, type:

```
! any shell command
!
```

When the command finishes, ed will signal you by displaying another !; at that point, you can resume editing.

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

4.8. 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. This section gives some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. For more information on each, refer to the *SunOS Reference Manual*.

Editing Scripts

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

For example, suppose you want to change every 'SUN' to 'Sun' and every 'SYSTEM' to 'System' in a large number of files. Then put into a file, which we'll call *changes*, the lines:

```
g/SUN/s//Sun/g
g/SYSTEM/s//System/g
w
q
```

Now you can say:

```
hostname% ed file1 <script
hostname% ed file2 <script
...
```

This causes ed to take its commands from the prepared script called *changes*. Notice that you have to plan the whole job in advance.

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

Matching Patterns with 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. You can 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 gets around these limitations. The search patterns that are described in this chapter are often called 'regular expressions', and 'grep' stands for 'general regular expression, print.' That describes exactly what grep does — it displays every line in a set of files that contains a particular pattern. Thus, to find 'thing' wherever it occurs in any of the files *file1*, *file2*, etc., type:

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

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 SunOS 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 *do not* contain a pattern:

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

finds all lines that don't contain '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', use:

```
hostname% 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 the *SunOS User's Guide: Doing More* for an introduction to 'piping.' See the *SunOS Reference Manual* for details on grep.

4.9. 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 filename. 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, that is, add lines to the buffer at line dot, unless a different line is specified. Type a '.' on a new line to terminate appending. Dot is set to the last line appended.

c   Change the specified lines to the new text that follows. Type a '.' as with a to terminate the change. If no lines are specified, replace line dot. Dot is set to last line changed.

d   Delete the lines specified. If none is 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 use 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
```

executes 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   Display specified lines. If none is specified, display line dot. A single line number is equivalent to *line-number* p. Type a single ⌈RETURN⌉ to show . +1, the next line.

q   Quit ed. This wipes out all text in buffer if you give it twice in a row without first giving a w command.

r   Read a file into the buffer at the end unless an address is specified. Dot is set to the last line read.

s   The command:

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

substitutes the characters 'string2' into 'string1' 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. An s changes

sun microsystems

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 into a file. Dot is not changed.

.=   Show value of dot (current line number). An '=' by itself shows the value of '$.' (number of the last line in the buffer).

!   The line:

```
!command
```

executes *command* as a SunOS shell command.

/-----/ Context search. Search for next line that contains this string of characters and display 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 '$.'.

Using sed, the Stream Text Editor

SunOS provides a stream editor called sed that you can use to search through a file and edit it temporarily. sed is particularly useful for transient changes. sed commands can reside in a file or can be given on the command line. sed edits a file non-interactively and prints out the edited lines on standard output. The actual file remains unchanged and the changes are not saved permanently unless you redirect the sed output to a file.

5.1. Introduction

This chapter describes sed, the non-interactive context or *stream* editor.[17] Use sed for editing files too large for comfortable interactive editing, editing any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode, and performing multiple global editing functions efficiently in one pass through the input. Because the default mode is to apply edit commands globally, and because its output is to the standard output, your workstation or terminal screen, sed is good for making changes of a transient nature, rather than permanent modifications to a file.

You can create a complicated editing script separately and use it as a command file. For complex edits, this saves considerable typing, and its attendant errors. Running sed from a command file is much more efficient than any interactive editor even if that editor can be driven by a pre-written script.

Whereas the ed editor copies your original file into a buffer, sed does not use temporary files so you can edit any size file. The only space requirement is that the input and output fit simultaneously into the available second storage. Additionally, ed lets you explore the text in whatever order you want, while sed works on your file from beginning to end, and allows you no choice of edit commands once you have started it. Basically sed passes some data through a set of transformations called editor *functions*.

By default sed copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. You can modify this behavior by adding a command-line option; see the "Command Options" section below.

---

[17] The material in this chapter is derived from *Sed — a Non-Interactive Text Editor*, L.E. McMahon, Bell Laboratories, Murray Hill, New Jersey.

As a lineal descendant of the ed editor, sed recognizes basically the same regular expressions as ed. The range of pattern matches is called the *pattern space*. Ordinarily, the pattern space is one line of text, but you can read more than one line into the pattern space if necessary. But because of the differences between interactive and non-interactive operation, ed and sed are different enough that even experienced ed users should read this chapter. You cannot use relative addressing with sed as you can with an interactive editor because sed operates a line at a time. sed also does not give you any immediate verification that a command has done what was intended.

Refer to the chapter on "Using the ed Line Editor" in *Editing Text Files on the Sun Workstation* for more information on ed and to the man pages for sed and ed in *SunOS Reference Manual*.

5.2. Using sed

The general format of an editing command is:

```
sed [line1[,line2]] function [arguments]
```

There is an optional line address, or two line addresses separated by a comma, a single-letter edit function, followed by other arguments, which may be required or optional, depending on which function you use. See the section "Specifying Lines for Editing" for the format of line addresses. Any number of blanks or tabs may separate the line addresses from the function. sed ignores tab characters and spaces at the beginning of lines. The function must be present; the available commands are discussed in the "Functions" section under each individual function name. You can either put the edit commands on the sed command line or put the commands in a file, which is then applied to the file you want to edit. If the commands are few and simple, put them on the sed command line. For example, assume the following input text in a file called *kubla*:

```
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.
```

Let's copy the first two lines of input as a simple example:

```
hostname% sed 2q kubla
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

As another example, suppose that you want to change the 'Khan' to 'KHAN.' Then the command:

```
hostname% sed  s/Khan/KHAN/g kubla
```

applies the command 's/Khan/KAN/' to all lines from *kubla* and copies all lines

to the standard output. The advantage of using sed in such a case is that you can use it 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:

```
hostname% sed  -f   cmdfile input-files...
```

Command Options

sed has three options that modify sed's action. If you invoke sed with the −f (file) option, the edit commands are taken from a file. For example:

```
hostname% sed -f edcomds oldfile > newfile
```

The name of the file containing the edit commands must immediately follow the −f option. Here, the edit commands in the *edcomds* file are applied to the file *oldfile*, and the standard output is redirected to *newfile*.

You use the −e (edit) option to place editing commands directly on the sed command line. If you are only using one edit command, you can omit the −e, but we include it in the example below for instructive purposes. For example, to delete a line containing the string 'Khan' from *kubla*, you type:

```
hostname% sed -e /Khan/d kubla > newkubla
```

If you put more than one edit command on the sed command line, each one must be preceded by −e. For example:

```
hostname% sed -e /Khan/d -e s/decree/DECREE/ newkubla
```

You can also use both the −e and the −f options at the same time.

sed normally copies all input lines that are changed by the edit operation to the output. If you want to suppress this normal output, and have only specific lines appear on the output, use the −n option with the p (print) flag. For example:

```
hostname% sed -n -e s/to/by/p kubla
Through caverns measureless by man
Down by a sunless sea.
```

As a quick reference, these options are:

−f   Use the next argument as a filename; the file should contain one editing command to a line.

−e	Use the next argument as an editing command.
−n	Send only those lines to the output specified by p functions or p functions after substitute functions (see the "Input-Output Functions" section).

5.3. Editing Commands Application Order

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a moderately efficient form for execution 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.

You can change the default linear order of application of editing commands by the flow-of-control commands, t and b (see the "Flow-of-Control Functions" section). Even when you change the order of application by these commands, it is still true that the input line to any command is the output of any previously applied command.

5.4. Specifying Lines for Editing

Use addresses to select lines in the input file(s) to apply the editing commands to. Addresses may be either line numbers or context addresses.

Group one address or address-pair with curly braces '{ }' to control the application of a group of commands. See the "Flow-of-Control Functions" section for more on this.

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 or '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.

Context Addresses

A context address is a pattern or *regular expression* enclosed in slashes (/). sed recognizes the regular expressions that are constructed as follows:

ordinary character

An ordinary character (not one of those discussed below) is a regular expression, and matches that character.

^   A circumflex ^ at the beginning of a regular expression matches the null character at the beginning of a line.

$ A dollar-sign $ at the end of a regular expression matches the null character at the end of a line.

\n   The characters backslash and en \n match an embedded newline character, but not the newline at the end of the pattern space.

.   A period . matches any character except the terminal newline of the pattern space.

*   A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.

[*character string*]

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 a circumflex ^, the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.

concatenation

A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.

\(\)   A regular expression between the sequences \(and \) is identical in effect to the unadorned regular expression, but has side-effects which are described in the section entitled "The Substitute Function s" and immediately below.

\d   This stands for 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.

null   The null regular expression standing alone (such as, / /) is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal, that is, 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.

Number of Addresses

The commands described in the "Functions" section can have 0, 1, or 2 addresses. Specifying more than the maximum number of addresses allowed is 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 that match that address. If a command has two addresses, it is applied to the inclusive range defined by those two addresses.

The command is applied to the first line that 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. A comma separates two addresses.

For example:

```
/an/                          matches lines 1, 3, 4 in our sample kubla file
In Xanadu did Kubla Khan
Where Alph, the sacred river, ran
Through caverns measureless to man

/an.*an/                      matches line 1
In Xanadu did Kubla Khan

/^an/                         matches no lines


/./                           matches all lines
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.

/\./                          matches line 5
Down to a sunless sea.

/r*an/                        matches lines 1,3, 4 (number = zero!)
In Xanadu did Kubla Khan
Where Alph, the sacred river, ran
Through caverns measureless to man

/\(an\).*\1/                  matches line 1
In Xanadu did Kubla Khan
```

5.5. Functions

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is enclosed in parentheses, followed by the single character function name and possible arguments in italics. The summary provides an expanded English translation of the single-character name, and a description of what each function does.

Whole-Line Functions

The functions that operate on a whole line of input text are as follows:

(2) **d**   Delete lines. The d function deletes from the file all those lines matched by its address(es); that is, it does not write the indicated lines to the output, No further commands are attempted on 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. For example, this command prints just the head of a file, by deleting all lines after the 10th line:

```
hostname% sed '11,$d' filename
```

(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 writes the argument *text* to the output after the line matched by its address. The a function 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, 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 that 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 change the line-number counter. For example, this sed script appends a .LP after every .H header:

```
/^.H /a\
.LP
```

(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. For example, this sed script inserts a need command before every .FN macro:

```
/^.FN /i\
.br\
.ne 2i
```

(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, put a newline hidden by a backslash after c; interior new lines in *text* must also be hidden by backslashes. The c function 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. For example, this sed script changes all .DS macro lines to a .BS and a .LS macro:

```
        /^.DS/c\
        .BS\
        .LS
```

> No further commands are attempted on a line deleted by a c function. 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. See the section "Multiple Input-line Functions" later in this chapter for a description of the r function.

Note: Leading blanks and tabs are not displayed in the output produced by these functions. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash does not appear in the output.

For example, put the following list of editing commands in a file called *Xkubla*:

```
hostname% cat > Xkubla
n
a\
XXXX
d
^D
hostname% sed -f Xkubla kubla
In Xanadu did Kubla 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
i\
XXXX
d
```

or

```
n
c\
XXXX
```

The Substitute Function s

The s (substitute) function changes parts of lines selected by a context search within the line. The standard format is the same as the ed substitute command:

> *(2)* s *pattern replacement flags*

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 described in the "Specifying Lines for Editing" section. The only difference between *pattern* and a context address is that the context address must be delimited by slash (/) characters; you can delimit *pattern* by any character other than space or newline.

By default, only the first string matched by *pattern* is replaced. 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   Is replaced by the *d*th substring matched by parts of *pattern* enclosed in \\ (and \\ ) where *d* is a single digit. If nested substrings occur in *pattern*, the *d*th is determined by counting opening delimiters ('\\(').

As in patterns, you can make the special characters (&, +, and \\) literal by preceding them with a 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 or 'display' the line if a successful replacement was done. The p flag writes the line 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 writes lines which are actually substituted by the s function 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. You can specify a
maximum of 10 different filenames after w flags and w functions (see
below), combined.

For example, applying the following command to the *kubla* file produces on the
standard output:

```
hostname% sed -e "s/to/by/w changes" kubla
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

Note that if the edit command contains spaces, you must enclose it with quotes.

It also creates a new file called *changes* that contains only the lines changed as
you can see using the more command:

```
hostname% more changes
Through caverns measureless by man
Down by a sunless sea.
```

If the nocopy option −n is in effect, you see those lines that are changed:

```
hostname% sed -e "s/[.,;?:]/*P&*/gp" -n kubla
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 assuming nocopy mode, consider:

```
hostname% sed -e "/X/s/an/AN/p" -n kubla
In XANadu did Kubla Khan
```

and the command:

```
hostname% sed -e "/X/s/an/AN/gp" -n kubla
In XANadu did Kubla KhAN
```

Input-output Functions

The following functions affect the input and output of text. The maximum
number of allowable addresses is in parentheses.

(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 encoun-
               tered, regardless of what succeeding editing commands may do to
               the lines.

(2) w *filename*

> Write to *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. Put only one space between w and *filename*. You can use a maximum of ten different files in write functions and with 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 you execute r and a functions 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. Put only one space between 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 displayed.

Note: Since there is a limit to the number of files that can be opened simultaneously, put no more than ten files in w functions or flags; reduce that number by one if any r functions are present. Only one read file is open at one time.

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 reads in *note1* after the line containing 'Kubla':

```
hostname% sed -e "/Kubla/r note1" kubla
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.
```

Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with *pattern spaces* containing embedded newlines; they are intended principally to provide pattern matches across lines in the input. A pattern space is the range of pattern matches. 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 function described below.

The maximum number of allowable addresses is enclosed in parentheses.

(2) N   Next line. The next input line is appended to the current line in the pattern space; an embedded newline separates the two input lines. Pattern matches may extend across the embedded 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 or 'display' 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 embedded newlines in the pattern space.

Hold and Get Functions   Four functions save and retrieve part of the input for possible later use.

(2) h   Hold pattern space. The h function 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.

For example, if you want to add : In Xanadu to our standard example, create a file called *test* containing the following commands:

```
1h
1s/ did.*//
1x
G
s/\n/   :/
```

Then run that file on the *kubla* file:

```
hostname% sed -f test kubla
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
```

Flow-of-Control Functions

These functions do not edit the input lines, but control the application of functions to the lines that are addressed.

(2) !　　　Called 'Don't', the ' ! ' function applies the next command, written on the same line, to all and only those input lines *not* selected by the address part.

(2) {　　　Grouping. The grouping command ' { ' applies (or does not apply) the next set of commands as a block to the input lines that the addresses of the grouping command select. The first of the commands under control of the grouping command may appear on the same line as the { or on the next line.

　　　A matching } standing on a line by itself terminates the group of commands. 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 restarts the sequence of editing commands being applied to the current input line 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. Either reading a new input line or executing a t function resets the flag which indicates that a successful substitution has occurred.

Miscellaneous Functions   Two additional functions are:

(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 writes the current line to the output if it
                 should be, writes any appended or read text, and terminates execu-
                 tion.

Scanning Files

This chapter describes different methods for scanning files. It is divided into two sections: the first on commands to view the contents of files, the second on commands to search though files.

6.1. Viewing Files

Viewing a file is much like editing, except you don't change the file when viewing it — you just want to look at it on the screen.

Seeing the Top with head

The head command displays the first 10 lines of a file. You can view a different number of lines by giving a numeric argument to head:

```
hostname% head -1 filename
```

This command shows just the first line of a file. If you give head a list of files, it shows the first few lines of each file in order:

```
hostname% head chap*
```

This shows the first ten lines of each file beginning with chap, including a label before each file.

Seeing the Bottom with tail

The tail command displays the last 10 lines of a file. You can view a different number of lines by giving a numeric argument to tail:

```
hostname% tail -1 filename
```

This command shows just the last line of a file. If you give tail a list of files, it shows only the last few lines of the final file. In order to look at the last lines of each file in a list, use the foreach construct of the C shell:

```
hostname% foreach file (chap*)
? echo === $file ===
? tail $file
? end
```

If you are running the Bourne shell, use the for construct instead.

Sequential Views with `cat`	For reading a file or a set of files from beginning to end, `cat` is the most efficient command. Its name is shorthand for *concatenate*. Note: the output scrolls too fast unless you enable page mode inside `sunview`, or have a terminal with scrolling control.
Selected Views with `more`	For people using ordinary terminals instead of Sun Workstations, `more` is a useful command, because it pauses after one screenful with a `--More--` message. This command also performs underlining. It also permits you to search forward for a pattern, and to return to the beginning of a file (but not a pipe). Unfortunately, `more` is not very efficient.
Random Views with `view`	This command invokes `vi` with the `readonly` option set, which means you can't accidentally overwrite the file. If you're accustomed to `vi` this is great, since you can move around the file easily, both backwards and forward. For all but the largest files, `view` is faster than `more`.

6.2. Searching Through Files

Often you need to search through files to find a string or operate on that string, or both. SunOS provides several different text utilities that approach the problem from several different angles.

The most sophisticated approach is taken by a program called `awk`, discussed in the next chapter. This program searches for a pattern (a string of characters) in a file and performs a specified action on the pattern. Actually `awk` is a programming language, so it is very flexible.

There is also a utility for searching for patterns and displaying them (usually on the standard output). This program, called `grep`, doesn't perform any operations on the pattern. To search for a pattern in a file or files with `grep` and perform an operation on the pattern, you would need to pipe the output from `grep` to another program. If you specify more than one input file for `grep` to search, `grep` precedes each line that matches the pattern with the name of the file that it came from.

There are two variations on `grep` that have similar functions: `egrep` and `fgrep`. `egrep` finds full regular expressions and `fgrep` searches only for fixed strings.

For looking up strings of characters quickly in a dictionary file like `/usr/dict/words`, there is the utility `look`. `look` behaves just like `grep` but unless you give `look` a different input file, it searches through a specific sorted file and prints out all lines that begin with *string*.

To search through a file and reverse the order of characters on every line, use the program `rev`.

A stream editor called `sed` is available, which permits you to search through a file and edit it temporarily. `sed` is particularly useful for transient changes. `sed` commands can reside in a file or can be given on the command line. `sed` edits a file non-interactively and prints out the edited lines on the standard output. The actual file remains unchanged and the changes are not saved permanently unless you redirect the `sed` output to a file.

The last text utility we present here, `wc`, searches through your input file and counts the number of lines, words, and characters.

Searches with `grep`

There are many occasions when you will want to determine which file contains something you are looking for, or whether a particular string of characters exists in any of a number of files. One of the most useful text utilities is `grep`. `grep` stands for 'global regular expression printer', a mouthful of non-mnemonic syllables. However, it is a very useful tool for searching through one or many files for a string of characters.

The synopsis of the `grep` command and its two related commands:

```
grep [-v] [-c] [-l] [-n] [-b] [-i] [-s] [-h] [-w]
     [-eexpression] expression [filename ...]

egrep [-v] [-c] [-l] [-n] [-b] [-s] [-h]
     [-eexpression] [-ffile] [expression] [filename] ... ]

fgrep [-v] [-x] [-c] [-l] [-n] [-b] [-i] [-s] [-h]
     [-eexpression] [-ffile] [strings] [filename] ... ]
```

`grep` is a utility program that searches a file or files for lines that contain strings of a specified pattern. When `grep` finds the lines that match the pattern, it prints them out on the standard output.

The two variations on `grep`, `egrep` and `fgrep`, have functions similar to `grep`. `egrep` finds full regular expressions and `fgrep` searches only for fixed strings. In general, `egrep` is the fastest of these programs. We will explain these two commands later in this section.

The simplest form of `grep` searches for a pattern that consists of a fixed character string. `grep`'s power lies in its ability to describe more complex patterns, called regular expressions.

Searching for Character Strings

`grep` in its simplest form looks for a fixed character string. For example, if you are trying to discover if a specific word exists in a file, you use the form `grep` *word file*. An example of the command, using the same input files as in the earlier example, is:

```
hostname% grep Linda women
Linda
```

This command searches for the string 'Linda' in the file 'women'. Since the `grep` command uses spaces to separate arguments on the command line, you have to be careful what you tell `grep` to search for. If the string you want to search for contains spaces or tabs, you must surround the string with some kind of delimiter like quotation marks (single or double). Another example:

```
hostname% grep 'Larry G' all
Larry G
```

This command searches for the string 'Larry G' in the file 'all'. Because the string 'Larry G' contains a space, we used single quotes to delimit the second argument to `grep`.

When any of the `grep` utilities is applied to more than one input file, the name of the file is displayed preceding each line that matches the pattern. For example:

```
hostname% grep Linda women all
women:Linda
all:Linda
```

This command searches through the two files 'women' and 'all' for the string 'Linda'. `grep` displays the names of the files in which it found the string.

Inverted Search for 'Everything Except'

`grep` has an option to print every line *except* those that match *string*. This is done with the −v option. An example would be:

```
hostname% grep -v "chicken soup" recipes.file
```

if you wanted to list the titles of your recipes to decide what to have for dinner, knowing only that you didn't want chicken soup. This command will print out everything except the line containing the string chicken soup.

Regular Expressions

Many times you can't exactly remember the entire string you want to find. You might remember how it begins, or how it ends, or some other feature. Or, you might want to perform some operation on every occurrence of a particular string in a particular position on a line in the file. You should take advantage of `grep`'s powerful feature of searching for regular expressions in text.

You can ask for patterns like '...all six-letter words starting with 'st'', or '...all strings looking like `.IP` and at the beginning of a line'.

Such a pattern or template is called a 'regular expression'. Regular expressions are possible because certain characters have special meanings. These characters are often called 'metacharacters' because they represent something other than their literal meaning.

Take care when using the characters $, *, [, ^, |, (,), and \, in the regular expression as these characters are also meaningful to the Bourne and C shells. Enclose the entire expression argument in single quotes (' ') to avoid having the shell interpret the metacharacter. Double quotes will work most of the time also.

Match Beginning and End of Line

Two of the simplest metacharacters to use are the caret (^) and the dollar sign ($). These match the beginning and end of a line, respectively. For example:

```
hostname% grep 'panic' file
```

matches any occurrence of the word 'panic' in the file *file*. But if you slightly

alter the command to:

```
hostname% grep '^panic' file
```

you will locate only occurrences of the word 'panic' at the beginnings of lines. Similarly, $ appearing at the end of a string matches the end of a line:

```
hostname% grep 'panic$' file
```

This last example will find only those occurrences of the word 'panic' that fall at the ends of lines.

Logically, you can specify with:

```
hostname% grep '^Do not push the panic button.$' file
```

because of the beginning-of-line and end-of-line match requirement, that you find only lines that consist entirely of this pattern and nothing else. Blank lines can be matched with the pattern ^$. If there are spaces or tabs or other non-printing characters on the line, the ^$ pattern will not match such lines.

A text pattern that matches at a specific place on a line is called an 'anchored match' because it is anchored to a specific position. The ^ and $ characters lose their special meanings if they appear in places other than the beginning of the pattern, or the end of the pattern, respectively.

Match Any Character

The period, or dot character, as often known, is a metacharacter that matches any character at all. So the string " st.... " selects all words beginning with 'st' and having four other characters, provided the word is preceded and followed by a space. To find such words at the beginning of a line, you use

```
hostname% grep '^st....' file
```

or the end of a line

```
hostname% grep 'st....$' file
```

What grep really finds is not only words starting with 'st', but any string of six characters starting with 'st' and preceded by a space. So

```
hostname% grep ' st.... ' file
```

finds any of the patterns:

> string st[10]
> starti stop-g
> search   story!

Specifying that you only want to search for letters is possible with character classes explained in the next section. Text patterns never match across lines; they only match within a line. This is because the dot metacharacter never matches a newline character.

Character Classes

Characters enclosed in brackets (`[  ]`) specify a set of characters that `grep` is to search for. The match is on any one of the characters inside the brackets. For example:

```
hostname% grep [Tt]his file
```

finds both 'this' and 'This'. The expression `^[abcxyz]` finds all lines beginning with 'a' or 'b' or 'c' or 'x' or 'y' or 'z'. Inside square brackets, the hyphen character (–) specifies a range of characters. The patterns:

`[a-z]`   all lower-case letters
`[A-Z]`   all upper-case letters
`[0-9]`   all digits

are very common regular expressions. So, in the previous example of words beginning with 'st', to really limit the search to letters, we could specify:

```
hostname% grep ' st[a-z][a-z][a-z][a-z] ' file
```

If the caret character (`^`) is the first character inside the square brackets, it does not mean 'beginning of line' anymore. Instead, it means anything *except* the search string. For example, the pattern:

```
hostname% grep ^[^a-z] file
```

finds all lines *except* those beginning with lower-case letters.

Note that ranges of letters refer to the ASCII character set so the range `[A-z]` not only finds all upper- and lower-case letters, but also all the other characters that fall in that range of ASCII character values, namely:

```
[      \      ]      ^      _      `
```

There are a few pitfalls you can avoid by paying close attention to syntax in specifying ranges of characters. For example, the pattern:

```
[1-30]
```

does *not* mean 'numbers in the range 1 through 30'. It means 'digits in the range 1 through 3, OR 0'. This is the same as specifying the pattern:

```
[1230]
```

or

```
[0-3]
```

If you want to include the hyphen character (–) in the class of characters, you just need to ensure that it won't be confused with a range specification. For example, a hyphen at the beginning of the pattern stands for itself:

```
[-ab]
```

This example means the pattern '–' or 'a' or 'b'. You should threat the characters [and] with this same caution.

Closures — Repeated Pattern Matches

A number enclosed in braces { } following an expression specifies the number of times the preceding expression is to be repeated. For example, in the earlier search for six-letter words beginning with 'st' could be expressed:

```
' st[a-z]{4} '
```

This repeat number specification is known as a 'closure'. The general format of the closure is *{n, m}, where n is the minimum number of repeats and m is assumed to be infinity (or at least huge). There are shorthand ways of expressing some closures:*

asterisk *   is equivalent to {0, },
                   meaning the preceding
                   pattern is to be repeated
                   zero or more times.

plus sign +   is equivalent to {1, },
                   meaning the preceding
                   pattern is to be repeated
                   one or more times.

question mark ?   is the same as {0,1},
                   which means that the
                   preceding pattern can be
                   repeated zero or once only.

Closures are the reason that text patterns do not span across lines. If you just type a grep pattern like this:

```
hostname% grep '.*' file
```

the pattern is trying to specify 'match zero to infinity amounts of any character'. If patterns could span lines, this would try to digest an entire file. Like any other

utility, `grep` has some limit to the size of the pattern it can hold internally. A whole file could be too large for `grep`.

Since patterns can not match a newline, the `grep` `'.*'` command in the example above finds and displays every line in *file*.

Searching for Fixed Strings — fgrep

The `fgrep` utility is another text processing utility in the same family as `grep` and `egrep` (described in the following section). The `fgrep` command only handles fixed character strings as text patterns. The `grep` command cannot process wild-card matches, character classes, anchored matches, or closures. For these reasons, `fgrep` is faster than `grep` when all you want to search for is a fixed character string.

An example of `fgrep` usage:

```
hostname% fgrep 'comma in' awk.msun
Items separated by a comma in the print statement
```

You can also give `fgrep` a file of fixed strings. Each string appears on a line by itself, but the newline characters have to be escaped with the backslash character (\).

Extended Regular Expressions — egrep

Another variation on the basic `grep` utility is `egrep`. `egrep` stands for 'extended `grep`'. The `egrep` command is an extension to the basic `grep` to allow full regular expressions.

`egrep` can handle more complex regular expressions, of the form: 'find a pattern, followed by this or that or one of those, followed by something else'. Alternative patterns are specified by separating the alternative patterns with the | (vertical bar) character. This form of regular expression is technically called 'alternation'.

Alternate patterns within regular expressions can be grouped by enclosing the patterns within parentheses ( ). For example:

```
hostname% egrep 'Roman (type|font)' font.change
This paragraph might appear in either Roman font or Italics
If this is Roman type, .LP resets the font; if Italic, .LP
```

In this example, `egrep` searches through the file font.change either for the string 'Roman type' or the string 'Roman font'. In the example, `egrep` found both so it printed two different lines each containing one of the patterns it searched for.

Note that the alternatives are in parentheses. If you had typed the command:

```
hostname% egrep 'Roman type|font' font.change
```

you would be searching for the strings 'Roman type' or 'font' and you would get a different result:

```
hostname% egrep 'Roman type|font' font.change
This paragraph might appear in either Roman font or Italics
depending on whether a .LP macro request resets the font.
If this is Roman type, .LP resets the font; if Italic, .LP
```

Here the first and second lines matched the pattern 'font' and the third line matched the pattern 'Roman type'.

There are other less-used options to `grep`, not covered in depth in this section, and they are summarized below.

Table 6-1   `grep` *Option Summary*

OPTIONS	
−v	Invert the search to only display lines that *do not* match.
−x	Display only those lines that match exactly — that is, only lines that match in their entirety (`fgrep` only).
−c	Display a count of matching lines.
−l	List once the names of files with matching lines separated by newlines.
−n	Precede each line by its relative line number in the file.
−b	Precede each line by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.
−i	Ignore the case of letters in making comparisons — that is, upper- and lower-case are considered identical. This applies to `grep` and `fgrep` only.
−s	Work silently, that is, display nothing except error messages. This is useful for checking the error status.
−w	Search for the expression as a word as if surrounded by '\<' and '\>' — `grep` only. (See `ex`).
−e *expression*	Same as a simple *expression* argument, but useful when the *expression* begins with a dash (−).
−f *file*	Take the regular expression (`egrep`) or string list (`fgrep`) from *file*.

Table 6-2   `grep` *Special Characters*

Characters
\	Escape character.[18] Backslash (\) followed by any single character other than newline matches that character.
^	Anchored match: matches the beginning of a line.
$	Anchored match: matches the end of a line.
.	Dot (or period). Matches any character.
c	Matches any single character not otherwise endowed with special meaning.
[*string*]	Character class: match any single character from *string*. Ranges of ASCII character codes may be abbreviated as in [a-z0-9]. A right-side square bracket (]) may occur only as the first character of the string. A literal − must be placed where it can't be mistaken as a range indicator. A caret (^) character immediately after the open bracket negates the sense of the character class, that is, the pattern matches any character *except* those in the character class.
*	Closure: a regular expression followed by an asterisk (*) matches a sequence of zero or more matches of the regular expression.
+	Closure: a regular expression followed by a plus (+) matches a sequence of one or more matches of the regular expression.
?	Closure: a regular expression followed by a question mark (?) matches a sequence of zero or one matches of the regular expression.
concatenation	Two regular expression concatenated match a match of the first followed by a match of the second.
|	Alternation: two regular expressions separated by a vertical bar (|) or newline match either a match for the first or a match for the second (`egrep` only).
()	A regular expression enclosed in parentheses matches a match for the regular expression.

---

[18] In this table, the term 'character' excludes newline.

The order of precedence of operators at the same parenthesis level is:

[ ]	character classes
* + —	closures
concatenation	
| and newline	alternation

Dictionary Search with look

For looking up strings of characters quickly in a dictionary file like /usr/dict/words, there is the utility look. look behaves just like grep but unless you give look a different input file, it searches through a specific sorted file and prints out all lines that begin with *string*.

look's function is to find lines in a sorted list. The synopsis of the look command is:

```
look  [-df]  string  [file]
```

The options to look are:

-d Dictionary order: only letters, digits, tabs and blanks participate in comparisons.

-f Fold: upper-case letters compare equal to lower-case.

If no file is specified, look uses /usr/dict/words with collating sequence —df.

Reversing Lines with rev

To look through a file and reverse the order of characters on every line, use the program rev. The synopsis of the rev command is:

```
rev  [file]  ...
```

rev copies the named files to the standard output, reversing the order of characters in every line. If no file is specified, the standard input is copied.

Counting Words with wc

The wc program searches through input files, counting the number of lines, words, and characters. The synopsis for the wc command is:

```
wc  [-lwc]  [file ... ]
```

wc counts lines, words, and characters in the named files, or in the standard input if no file names appear. A word is a string of characters delimited by spaces, tabs, or newlines.

If an argument beginning with one of the letters l, w, or c, is present, wc may:

l   Count lines.

w   Count words.

c   Count characters.

The default is to use all of the options in the order −lwc (count lines, words, and characters). Some examples are:

```
hostname% wc wc.1
      38      153       943 wc.1
hostname% wc -l wc.1
      38 wc.1
hostname% wc -w wc.1
     153 wc.1
hostname% wc -c wc.1
     943 wc.1
hostname% wc wc.1
     943 wc.1
hostname%
hostname% wc awk.1 grep.1 look.1 rev.1 sed.1
     224     1141      6713 awk.1
     246     1113      6548 grep.1
      22       95       614 look.1
      12       58       307 rev.1
     211     1053      6253 sed.1
     715     3460     20435 total
```

Pattern Scanning and Processing with
awk

awk is a utility program that you can program in varying degrees of complexity.
awk's basic operation is to search a set of files for patterns based on *selection
criteria*, and to perform specified actions on lines or groups of lines that contain
those patterns. Selection criteria can be text patterns or *regular expressions*.
awk makes data selection, transformation operations, information retrieval and
text manipulation easy to state and to perform.[19]

Basic awk operation is to scan a set of input lines in order, searching for lines
which match any of a set of patterns that you have specified. You can specify an
action to be performed on each line that matches the pattern.

awk patterns may include arbitrary Boolean combinations of regular expressions
and of relational and arithmetic operators on strings, numbers, fields, variables,
and array elements. Actions may include the same pattern-matching construc-
tions as in patterns, as well as arithmetic and string expressions and assignments,
if-else, while, for statements, and multiple output streams.

If you are familiar with the grep utility (see the *SunOS Reference Manual* for
details), you will recognize the approach, although in awk, the patterns may be
more general than in grep, and the actions allowed are more involved than
merely displaying the matching line.

As some simple examples to give you the idea, consider a short file called *sam-
ple*, which contains some identifying numbers and system names:

```
125.1303    krypton loghost
125.0x0733  window
125.1313    core
125.19      haley
```

If you want to display the second and first columns of information in that order,
use the awk program:

---

[19] The material in this chapter is derived from *Awk — A Pattern Scanning and Processing Language*, A.
Aho, B.W. Kernighan, P. Weinberger, Bell Laboratories, Murray Hill, New Jersey.

```
hostname% awk '{print $2, $1}' sample
krypton 125.1303
window 125.0x0733
core 125.1313
haley 125.19
```

This is good for reversing columns of tabular material for example. The next program shows all input lines with an a, b, or c in the second field.

```
hostname% awk '$2 ~ /a|b|c/' sample
125.1313    core
125.19      haley
```

7.1. Using awk

The general format for using awk follows. You execute the awk commands in a string that we'll call *program* on the set of named *files*:

```
hostname% awk   program files
```

For example, to display all input lines whose length exceeds 13 characters, use the program:

```
hostname% awk 'length > 13' sample
125.1303    krypton loghost
125.0x0733  window
```

In the above example, the *program* compares the length of the *sample* file lines to the number 13 and displays lines longer than 13 characters.

awk usually takes its program as the first argument. To take a program from a file instead, use the −f (file) option. For example, you can put the same statement in a file called *howlong*, and execute it on *sample* with:

```
hostname% awk  −f howlong   hosts
125.1303    krypton loghost
125.0x0733  window
```

You can also execute awk on the standard input if there are no files. Put single quotes around the awk program because the shell duplicates most of awk's special characters.

Program Structure

A program can consist of just an action to be performed on all lines in a file, as in the *howlong* example above. It can also contain a pattern that specifies the lines for the action to operate on. This pattern/action order is represented in awk notation by:

```
pattern    {action }
```

In other words, each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. Thus a line which matches several patterns can be printed several times. If there is no pattern for an action, the action is performed on every input line. A line which doesn't match any pattern is ignored. Since patterns and actions are both optional, you must enclose actions in braces ({action}) to distinguish them from patterns. See more about patterns in the "Specifying Patterns" section later in this chapter.

Records and Fields

awk input is divided into *records* terminated by a *record separator*. The default record separator is a newline, so by default awk processes its input a line at a time. The number of the current record is available in a variable named NR.

Each input record is considered to be divided into *fields*. Fields are separated by *field separators*, normally blanks or tabs, but you can change the input field separator, as described in the "Field Variables" section later in this chapter. Fields are referred to as $X where $1 is the first field, $2 the second, and so on as shown above. $0 is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named NF.

The variables FS and RS refer to the input field and record separators; you can change them at any time to any single character. You may also use the optional command-line argument −Fc to set FS to any character *c*.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable FILENAME contains the name of the current input file.

7.2. Displaying Text

The simplest action is to display (or *print*) some or all of a record with the awk command print. print copies the input to the output intact. An action without a pattern is executed for all lines. To display each record of the *sample* file, use:

```
hostname% awk '{print}' sample
125.1303    krypton loghost
125.0x0733  window
125.1313    core
125.19      haley
```

Remember to put single quotes around the awk program as we show here.

More useful than the above example is to print a field or fields from each record. For instance, to display the first two fields in reverse order, type:

```
hostname% awk '{print $2, $1}' sample
krypton 125.1303
window 125.0x0733
core 125.1313
```

Items separated by a comma in the `print` statement are separated by the current output field separator when output. Items not separated by commas are concatenated, so to run the first and second fields together, type:

```
hostname% awk '{print $1 $2}' sample
125.1303krypton
125.0x0733window
125.1313core
125.19haley
```

You can use the predefined variables NF and NR; for example, to print each record preceded by the record number and the number of fields, use:

```
hostname% awk '{ print NR, NF, $0 }' sample
1 3 125.1303    krypton loghost
2 2 125.0x0733  window
3 2 125.1313    core
4 2 125.19      haley
```

You may divert output to multiple files; the program:

```
hostname% awk '{print $1 >"foo1"; print $2 >"foo2"}' filename
```

writes the first field, $1, on the file *foo1*, and the second field on file *foo2*. You can also use the >> notation; to append the output to the file *foo* for example, say:

```
hostname% awk '{print $1 >>"foo"}' filename
```

In each case, the output files are created if necessary. The filename can be a variable or a field as well as a constant. For example, to use the contents of field 2 as a filename, type:

```
hostname% awk '{print $1 >$2}' filename
```

This program prints the contents of field 1 of *filename* on field 2. If you run this on our *sample* file, four new files are created. There is a limit of 10 output files.

Similarly, you can pipe output into another process. For instance, to mail the output of an `awk` program to *susan*, use:

```
hostname% awk '{ print NR, NF, $0 }' sample | mail susan
```

(See the *UNKNOWN TITLE ABBREVIATION: MMBG* for details on `mail`.)

To change the current output field separator and output record separator, use the variables `OFS` and `ORS`. The output record separator is appended to the output of the `print` statement.

`awk` also provides the `printf` statement for output formatting. To format the expressions in the list according to the specification in *format* and print them, use:

```
printf format, expr, expr, ...
```

To print `$1` as a floating point number eight digits wide, with two after the decimal point, and `$2` as a 10-digit long decimal number, followed by a newline, use:

```
hostname% awk '{printf("%8.2f  %10ld\n",$1,$2)}' filename
```

Notice that you have to specifically insert spaces or tab characters by enclosing them in quoted strings. Otherwise, the output appears all scrunched together. To print a double quote (`"`), precede it with a backslash. The version of `printf` is identical to that provided in the C Standard I/O library (see *printf* in *C Library Standard I/O* (3S) in the *SunOS Reference Manual*.

7.3. Specifying Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. You may use a variety of expressions as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary Boolean combinations of these.

BEGIN **and** END

`awk` has two built-in patterns, `BEGIN` and `END`. `BEGIN` matches the beginning of the input, before the first record is read. The pattern `END` matches the end of the input, after the last record has been processed. `BEGIN` and `END` thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by:

```
BEGIN   { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by:

```
END   { print NR }
```

If `BEGIN` is present, it must be the first pattern; `END` must be the last if used.

Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete awk program which displays all lines which contain any occurrence of the name 'smith'. If a line contains 'smith' as part of a larger word, it is also displayed. Suppose you have a file *testfile* that contains:

```
summertime
smith
blacksmithing
Smithsonian
hammersmith
```

If you use awk on it, the display is:

```
hostname% awk /smith/ testfile
smith
blacksmithing
hammersmith
```

awk regular expressions include the regular expression forms found in the text editor ed and in grep. In addition, awk uses parentheses for grouping, | for alternatives, + for 'one or more', and ? for 'zero or one', all as in lex. Character classes may be abbreviated. For example:

```
/[a-zA-Z0-9]/
```

is the set of all letters and digits. As an example, to display all lines which contain any of the names 'Adams,' 'West' or 'Smith,' whether capitalized or not, use:

```
'/[Aa]dams|[Ww]est|[Ss]mith/'
```

Enclose regular expressions (with the extensions listed above) in slashes, just as in ed and sed. For example:

```
hostname% awk '/[Ss]mith/' testfile
smith
blacksmithing
Smithsonian
hammersmith
```

finds both 'smith' and 'Smith'.

Within a regular expression, blanks and the regular expression metacharacters are significant. To turn off the magic meaning of one of the regular expression

characters, precede it with a backslash. An example is the pattern

```
/ \/ . * \//
```

which matches any string of characters enclosed in slashes.

Use the operators ~ and ! ~ to find if any field or variable matches a regular expression (or does not match it). The program

```
$1 ~ /[sS]mith/
```

displays all lines where the first field matches 'smith' or 'Smith.' Notice that this will also match 'blacksmithing', 'Smithsonian', and so on. To restrict it to exactly [sS]mith, use:

```
hostname% awk '$1 ~ /^[sS]mith$/' testfile
smith
```

The caret ^ refers to the beginning of a line or field; the dollar sign $ refers to the end.

Relational Expressions

An awk pattern can be a relational expression involving the usual relational and arithmetic operators <, <=, ==, !=, >=, and >, the same as those in C. An example is:

```
'$2 > $1 + 100'
```

which selects lines where the second field is at least 100 greater than the first field.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
hostname% awk '$1 >= "s"' testfile
smith
```

selects lines that begin with an 's', 't', 'u', etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

performs a string comparison between field 1 and field 2.

Combinations of Patterns

A pattern can be any Boolean combination of patterns, using the operators `||` (or), `&&` (and), and `!` (not). For example, to select lines where the first field begins with 's', but is not 'smith', use:

```
hostname% awk '$1 >= "s" && $1 < "t" && $1 != "smith"' testfile
summertime
```

`&&` and `||` guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

The program:

```
$1 !=prev {print; prev=$1}
```

displays all lines in which the first field is different from the previous first field.

Pattern Ranges

The pattern that selects an action may also consist of two patterns separated by a comma, as in

```
pattern1, pattern2   { ... }
```

In this case, the action is performed for each line between an occurrence of *pattern1* and the next occurrence of *pattern2* inclusive. For example, to display all lines between the strings 'sum' and 'black', use:

```
hostname% awk '/sum/, /black/' testfile
summertime
smith
blacksmithing
```

while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

7.4. Actions

An `awk` action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

Assignments, Variables, and Expressions

The simplest action is an *assignment*. For example, you can assign 1 to the *variable x*:

```
x = 1
```

The '1' is a simple expression. `awk` variables can take on numeric (floating point) or string values according to context. In

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance, to assign 7 to *x*, use:

```
x = "3" + "4"
```

Strings that cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables other than built-ins are initialized to the null string, which has numerical value zero; this eliminates the need for most BEGIN sections. For example, the sums of the first two fields can be computed by:

```
    { s1 += $1; s2 += $2 }
END { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, /, and % (mod). For example:

```
NF % 2 == 0
```

displays lines with an even number of fields. To display all lines with an even number of fields, use:

```
NF % 2 == 0
```

The C increment + + and decrement − − operators are also available, and so are the assignment operators +=, −=, *=, /=, and %=.

An awk pattern can be a *conditional expression* as well as a simple expression as in the 'x = 1' assignment above. The operators listed above may all be used in expressions. An awk program with a conditional expression specifies conditional selection based on properties of the individual fields in the record.

Field Variables

Fields in awk share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to.

To replace the first field of each line by its logarithm, say:

```
{ $1 = log($1); print }
```

Thus you can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
    $3 = "too big"
  print
}
```

which replaces the third field by 'too big' when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is considered numeric or string depends on context; fields are treated as strings in ambiguous cases like:

```
if ($1 == $2) ...
```

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields. To split the string 's' into 'array[1]' ..., 'array[n]', use:

```
n = split(s, array, sep)
```

This returns the number of elements found. If the sep argument is provided, it is used as the field separator; otherwise FS is used as the separator.

String Concatenation

Strings may be concatenated. For example:

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a print statement,

```
print $1 " is " $2
```

prints the two fields separated by ' is '. Variables and numeric expressions may also appear in concatenations.

Built-In Functions

awk provides several *built-in* functions.

length Function

The length function computes the length of a string of characters. This program shows each record, preceded by its length:

```
hostname% awk '{print length, $0}' testfile
10 summertime
5 smith
13 blacksmithing
11 Smithsonian
11 hammersmith
```

length by itself is a 'pseudo-variable' that yields the length of the current record; length(argument) is a function which yields the length of its argument, as in the equivalent:

```
hostname% awk '{print length($0), $0}' testfile
10 summertime
5 smith
13 blacksmithing
11 Smithsonian
11 hammersmith
```

The argument may be any expression.

awk also provides the arithmetic functions sqrt, log, exp, and int, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

displays lines whose length is less than 10 or greater than 20.

substring Function

The function substr(s, m, n) produces the substring of *s* that begins at position *m* (origin 1) and is at most *n* characters long. If *n* is omitted, the substring goes to the end of *s*.

index Function

The function index(s1, s2) returns the position where the string *s2* occurs in *s1*, or zero if it does not.

sprintf Function

The function sprintf (*f*, *e1*, *e2*, ...) produces the value of the expressions *e1*, *e2*, and so on, in the printf format specified by *f*. Thus, for example, to set *x* to the string produced by formatting the values of $1 and $2, use:

```
x = sprintf("%8.2f %10ld", $1, $2)
```

Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the *NR -th* element of the array *x*. In fact, it is possible in principle though perhaps slow to process the entire input in a random order with the awk program

```
     { x[NR] = $0 }
END  { ... program ... }
```

The first action merely records each input line in the array x.

Array elements may be named by non-numeric values, which gives awk a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like 'apple', 'orange', etc. Then the program

```
/apple/     { x["apple"]++ }
/orange/    { x["orange"]++ }
END       { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

Flow-of-Control Statements

awk provides the basic flow-of-control statements if-else, while, for, and statement grouping with braces, as in C. We showed the if statement in the "Field Variables" section without describing it. The condition in parentheses is evaluated; if it is true, the statement following the if is done. The else part is optional.

The while statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
    print $i
    ++i
}
```

The for statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
    print $i
```

does the same job as the while statement above.

There is an alternate form of the `for` statement which is suited for accessing the elements of an associative array:

```
for (i in array)
    statement
```

does *statement* with *i* set in turn to each element of *array*. The elements are accessed in an apparently random order. Chaos will ensue if *i* is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an `if`, `while` or `for` can include relational operators like <, <=, >, >=, == ('is equal to'), and != ('not equal to'); regular expression matches with the match operators ~ and !~; the logical operators | |, &&, and !; and of course parentheses for grouping.

The `break` statement causes an immediate exit from an enclosing `while` or `for`; the `continue` statement causes the next iteration to begin.

The `next` statement causes `awk` to skip immediately to the next input record and begin scanning the patterns from the top of program. The `exit` statement causes the program to behave as if the end of the input had occurred.

You may put comments in `awk` programs: begin them with the character # and end them with the end of the line, as in

```
print x, y   # this is a comment
```

The `getline` Function

The `getline` function fetches the next input record and performs the normal field-splitting operations on it, setting NF and NR. It returns 1 if there was a record present, and 0 if end-of-file was encountered. Control flow continues at that point of the program.

Manipulating Files

8.1. Comparing Different Files

Occasionally you want to know whether two files are identical, or if they are not, what the differences are. There exist several different text utilities for comparing the contents of files. You can choose the command best for the task at hand, based on what kind of information it conveys to you. Most of the commands issue no output if the files are the same. Some return terse output stating barely more than the fact that the files differ. Others give a more complete summary of how the files differ and how you would have to modify one file to match the other(s).

The command cmp is an example of a command that issues terse output. At most, cmp prints the byte and line number where the files differ. Two other functions for directly comparing files are diff and comm. comm compares two files, putting the comparison information into three different columns: column one lists lines only in *file1*, column two lists lines only in *file2*, and column three lists lines common to both files. diff compares files and also directories. A special version of diff, diff3, also compares three files, identifying the differing contents with special flags.

The relational database operator join compares a specific field or fields in two files. Each time join finds the compared fields in the two files identical, it produces one output line.

For comparing adjacent lines in a single file, there is the command uniq. uniq can be made to report merely the repeated lines or to count them or to remove all but the first occurrence.

Comparing Binaries with cmp

The command cmp is for comparing two files. The synopsis of the cmp command is:

```
cmp [-l] [-s] file1 file2
```

cmp compares *file1* and *file2*. If *file1* is the standard input ('−'), cmp reads from the standard input. Under default options, cmp makes no comment if the files are the same. If the files differ, cmp announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

The options available with `cmp` are:

−l Print the byte number (decimal) and the differing bytes (octal) for each
    difference.

−s Print nothing for differing files; return codes only.

Comparing Text with `diff`   For summarizing the differences between two files or directories, `diff` is the
appropriate tool. To use the `diff` command, you would follow one of these
models:

```
diff [-cefh] [-b] file1 file2
diff [-Dstring] [-b] file1 file2
diff [-l] [-r] [-s] [] [-Sname] [-cefh] [-b] dir1 dir2
```

`diff` is a differential file comparator. When run on regular files, and when com-
paring text files that differ during directory comparison (see the notes below on
comparing directories), `diff` tells what lines must be changed in the files to
bring them into agreement. Except in rare circumstances, `diff` finds a smallest
sufficient set of file differences. If neither *file1* nor *file2* is a directory, either may
be given as '−', in which case the standard input is used. If *file1* is a directory, a
file in that directory whose file-name is the same as the file-name of *file2* is used
(and vice versa).

There are several options for output format; the default output format contains
lines of these forms:

```
n1 a n3,n4
n1,n2 d n3
n1,n2 c n3,n4
```

These lines resemble `ed` commands to convert *file1* into *file2*. The numbers after
the letters pertain to *file2*. In fact, by exchanging 'a' for 'd' and reading back-
ward you can see how to convert *file2* into *file1*. As in `ed`, identical pairs where
$n1 = n2$ or $n3 = n4$ are abbreviated as a single number.

Following each of these specification lines come all the lines that are affected in
the first file flagged by the character '<', then all the lines that are affected in the
second file flagged by the '>' character.

If both arguments are directories, `diff` sorts the contents of the directories by
name, and then runs the regular file `diff` program as described above on text
files that are different. Binary files that differ, common subdirectories, and files
that appear in only one directory are listed.

`diff` — First Form	To produce a script of append (a), change (c), and delete (d) commands for the editor `ed`, which will recreate *file2* from *file1*, use the first form of `diff` with the option −e.

Extra commands are added to the output when comparing directories with `diff` −e, so that the result is a Bourne shell (sh) script for converting text files common to the two directories from their state in *dir1* to their state in *dir2*.

To produce a script similar to that using −e, but in the opposite order, that is, to recreate *file1* from *file2*, use `diff` −f. The script generated with the −f option is not useful with `ed`, however.

To surround the specification lines the simplest use of `diff` puts out with some lines of context, use `diff` −c. The default is to present three lines of context. To change this (to 10, for example), add 10 to the −c option (-c10). With the −c option, the output format is slightly different from other `diff` output. It begins by identifying the files involved and the dates they were created. Then each change is separated by a line with a dozen stars (*). The lines removed from *file1* are marked with '−'; those added to *file2* are marked '+'. Lines that are changed from one file to the other are marked in both files with '!'.

If you know you've only made small changes to the files you are comparing, and you want to speed up the time `diff` takes to work, you can use `diff` −h. This command only does a fast, half-hearted job. `diff` −h works only when changed stretches are short and well-separated, but does work on files of unlimited length.

Except for the −b option, which my be given with any of the others, the options −c, −e, −f, and −h are mutually exclusive.

`diff` — Second Form	To create a merged version of *file1* and *file2* on the standard output with C preprocessor controls included, use the second form of `diff` with the option −D*string*. Compiling the result without defining *string* is equivalent to compiling *file1*, while compiling the result with *string* defined will yield *file2*.

If you want `diff` to ignore trailing blanks (spaces and tabs), use the option −b. Other strings of blanks compare equal. The way `diff` works, when it compares directories with the −b option specified, `diff` first compares the files (as in `cmp`), and then decides to run the `diff` algorithm if they are not equal. This may cause a small amount of spurious output if the files then turn out to be identical, because the only differences are insignificant blank string differences.

`diff` — Third Form	When comparing directories, you might be interested in several different things. If `diff` puts out a lot of output, you probably want to use the −l option (for long output). Each text file `diff` is piped through the program `pr` to paginate it, (see "Printing Files" later in this manual). Other differences are remembered and summarized after all text file differences are reported.

To compare directories and subdirectories, use the −r option. −r applies `diff` recursively to common subdirectories encountered.

Since `diff` ordinarily only outputs information on files and directories that differ, if a file or several files are identical in directories you are comparing, you won't see the identical files listed in the output. The −s option reports files that

are the same, in addition to the usual `diff` output, which are otherwise not mentioned.

Here are two directories, *macros* and *new*. For this example, here are lists of their contents.

```
hostname% ls macros
Makefile                making.index.msun       summary.msun
SunMacros.msun          mechanisms.msun         test.tr
contents.pic            mmemo.7                 text.effects.msun
contentsfile.msun       model.makefile.msun     troff.msun
document.styles.msun    process.pic
intro.msun              structures.msun
```

```
hostname% ls new
Makefile                making.index.msun       summary.msun
SunMacros.msun          mechanisms.msun         test.tr
contents.pic            mmemo.7                 text.effects.msun
contentsfile.msun       model.makefile.msun     troff.msun
document.styles.msun    process.pic
intro.msun              structures.msun
```

Right now these two directories are identical. The output of `diff` for these two directories *macros* and *new*, if there are no differences is:

```
hostname% diff macros new
```

The normal output is nothing, no response. Now if we edit some files and remove some others in the directory *new*, leaving the files like this:

```
hostname% ls macros new
macros:
Makefile                making.index.msun       summary.msun
SunMacros.msun          mechanisms.msun         test.tr
contents.pic            mmemo.7                 text.effects.msun
contentsfile.msun       model.makefile.msun     troff.msun
document.styles.msun    process.pic
intro.msun              structures.msun

new:
Makefile                intro.msun              structures.msun
SunMacros.msun          making.index.msun       summary.msun
contents.pic            mechanisms.msun         text.effects.msun
document.styles.msun    model.makefile.msun     troff.msun
```

The regular `diff` output looks like this:

```
hostname% diff macros new
diff macros/Makefile new/Makefile
7c7
< FORMATTER = /usr/local/iroff
---
> FORMATTER = /usr/doctools/bin/troff
Only in macros: contentsfile.msun
diff macros/intro.msun new/intro.msun
0a1
> .LP
6,10c7,9
< Document preparation at Sun Microsystems relies on variations of the
< .I troff
< text formatter as the underlying mechanism for turning your wishes into
< printed words and outlines on paper.  Using
< .I troff
---
> Document preparation at Sun Microsystems relies on variations of the
> troff text formatter as the underlying mechanism for turning your wishes
> into printed words and outlines on paper.  Using troff
Only in macros: mmemo.7
diff macros/model.makefile.msun new/model.makefile.msun
3,7c3
< The
< .I Makefile
< below is the
< .I Makefile
< used to actually make this document:
---
> The Makefile below is the Makefile used to actually make this document:
Only in macros: process.pic
Only in macros: test.tr
hostname%
```

The output of `diff` is rather cryptic. But if you look carefully at the specification lines and the direction of the angle brackets, you can decipher the results accurately.

To get a more complete picture of how the two directories compare, you might want to know which files are identical and which files exist only in one directory. For this, you use `diff -s`. The `diff -s` output from our example above looks like this:

```
hostname% diff -s macros new
diff -s macros/Makefile new/Makefile
7c7
< FORMATTER = /usr/local/iroff
---
> FORMATTER = /usr/doctools/bin/troff
Files macros/SunMacros.msun and new/SunMacros.msun are identical
Files macros/contents.pic and new/contents.pic are identical
Only in macros: contentsfile.msun
Files macros/document.styles.msun and new/document.styles.msun are identical
diff -s macros/intro.msun new/intro.msun
0a1
> .LP
6,10c7,9
< Document preparation at Sun Microsystems relies on variations of the
< .I troff
< text formatter as the underlying mechanism for turning your wishes into
< printed words and outlines on paper.  Using
< .I troff
---
> Document preparation at Sun Microsystems relies on variations of the
> troff text formatter as the underlying mechanism for turning your wishes
> into printed words and outlines on paper.  Using troff
Files macros/making.index.msun and new/making.index.msun are identical
Files macros/mechanisms.msun and new/mechanisms.msun are identical
Only in macros: mmemo.7
diff -s macros/model.makefile.msun new/model.makefile.msun
3,7c3
< The
< .I Makefile
< below is the
< .I Makefile
< used to actually make this document:
---
> The Makefile below is the Makefile used to actually make this document:
Only in macros: process.pic
Files macros/structures.msun and new/structures.msun are identical
Files macros/summary.msun and new/summary.msun are identical
Only in macros: test.tr
Files macros/text.effects.msun and new/text.effects.msun are identical
Files macros/troff.msun and new/troff.msun are identical
hostname%
```

To compare two directories beginning somewhere in the middle of the directories, use the option −S*filename* where *filename* is a file in one of the directories you are comparing. The syntax for this command is

```
diff −Sfilename dir1 dir2
```

For example, comparing the two directories from the example above, and

beginning with the file `model.makefile.msun`:

```
hostname% diff -Smodel.makefile.msun macros new
diff macros/model.makefile.msun new/model.makefile.msun
3,7c3
< The
< .I Makefile
< below is the
< .I Makefile
< used to actually make this document:
---
> The Makefile below is the Makefile used to actually make this document:
Only in macros: process.pic
Only in macros: test.tr
```

Three Files — `diff3`

If you have three versions of a file that you want to compare at once, use the `diff3` command. The synopsis for the `diff3` command is:

```
diff3 [-ex3] file1 file2 file3
```

`diff3` compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

`====`	all three files differ
`====1`	*file1* is different
`====2`	*file2* is different
`====3`	*file3* is different

The type of change required to convert a given range of a given file to a range in some other file is indicated in one of these ways:

```
f : n1 a
```

Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.

```
f : n1 , n2 c
```

Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a c indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the −e option, diff3 publishes a script for the editor ed that will incorporate into *file1* all changes between *file2* and *file3*, (that is, the changes that normally would be flagged ==== and ====3). Option −x produces a script to incorporate only changes flagged ====. Option −3 produces a script to incorporate only changes flagged ====3. The following command will apply the resulting script to *file1*.

```
(cat script; echo '1,$p') | ed - file1
```

Note: Text lines that consist of a single dot ('.') will defeat the −e option.

Table 8-1   diff3 *Option Summary*

OPTIONS
−e   Publish a script for the editor ed that will incorporate into *file1* all changes between *file2* and *file3*, (that is, the changes that normally would be flagged ==== and ====3).
−x   Produce a script for ed to incorporate only changes flagged ====.
−3   Produce a script for ed to incorporate only changes flagged ====3.

Finding Common Lines with comm

The comm command prints lines that are common to two files. comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence, but at least in the *same* order, and produces a three-column output:

Column 1   **Column 2**   **Column 3**
lines only in *file1*   lines only in *file2*   lines in both files

The synopsis of the comm command is:

```
comm [-[123]] file1 file2
```

As an example of the comm command's output, consider these files:

```
hostname% cat all
Aaron
Bruce
Dave
Elaine
Greg
Joe
Jon
Kevin
Larry G
Larry K
Linda
Mary
Mike B
Mike F
Niel
Pam
Randy
Sid
Tad
Tom
Wanda
hostname%
```

```
hostname% cat women
Christy
Cyndi
Elaine
Gale
Jeanette
Julia
Katherine
Katy
Linda
Lori
Mary
Pam
Pat
Patti
Rose Marie
Susan
Wanda
```

Here is the output of comm. The three columns overlap making output from files with long lines a little difficult to read.

```
hostname% comm women all
        Aaron
        Bruce
Christy
Cyndi
        Dave
                Elaine
Gale
        Greg
Jeanette
        Joe
        Jon
Julia
Katherine
Katy
        Kevin
        Larry G
        Larry K
                Linda
Lori
                Mary
        Mike B
        Mike F
        Niel
                Pam
Pat
Patti
        Randy
Rose Marie
        Sid
Susan
        Tad
        Tom
                Wanda
```

The filename '−' means the standard input. The flags 1, 2, or 3, suppress printing of the corresponding column. Thus:

```
hostname% comm −12
```

prints only the lines common to the two files, and

```
hostname% comm −23
```

prints only lines in the first file, but not in the second. (comm −123 does nothing).

Combining Files with `join`   To compare two files of database information and output a *join* of two fields, there is a utility called `join`. `join` is a relational database operator. The synopsis of the command is:

```
hostname% join [-an] [-e string] [-j[1|2] m] [-o list] [-tc] file1 file2
```

The program `join` forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is '–', the standard input is used.

file1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*. Fields are separated by blanks, tabs or newlines. Multiple separators count as one, and leading separators are discarded.

Note: With default field separation, the collating sequence is that of `sort –b`. Using the `join –t`, the sequence is that of a plain sort.

Table 8-2   `join` Option Summary

OPTIONS	
–a*n*	The parameter *n* can be one of the values: 1 produce a line for each unpairable line in *file1*. 2 produce a line for each unpairable line in *file2*. 3 produce a line for each unpairable line in both *file1* and *file2*. in addition to producing the normal output.
–e *string*	Replace empty output fields with *string*.
–j[1|2] *m*	Join on the *m*th field of file *n*, where *n* is 1 or 2. If *n* is missing, use the *m*th field in each file. Note that `join` counts fields from 1 instead of 0 like `sort` does.
–o *list*	Each output line comprises the fields specified in *list*, each element of which has the form *n.m*, where *n* is a file number and *m* is a field number.
–t*c*	Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

Repeated Lines and `uniq`

If you want to check your input file for repeated lines, use `uniq`, which reports repeated lines in a file.

The synopsis of the `uniq` command is:

```
uniq [-udc [+n] [-n] ]  [input file [output file] ]
```

`uniq` reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder of the text (no repeated lines) is written in the output file. Note that repeated lines must be adjacent in order to be found.

Normally, the lines in the input file that were not repeated and the first occurrence of the lines that were repeated forms the output. If you want to isolate either of these functions, you can specify either the −u or the −d option. `uniq` −u copies only the lines *not* repeated in the original file to the output file. `uniq` −d writes one copy of just the repeated lines to the output file.

In case you are interested in knowing how many occurrences of a given line appear in the input file, you can use the option `uniq` −c. With the −c option, you get first the number of occurrences, then the output in default format (all of the unique lines and no adjacent repeated lines).

There is also an option to compare the latter parts of lines rather than entire lines. The *n* arguments specify skipping an initial portion of each line in the comparison:

−n   The first *n* fields, together with any blanks before each, are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+n   The first *n* characters are ignored. Fields are skipped before characters.

Table 8-3   `uniq` *Option Summary*

OPTIONS	
−u	Copy only those lines that are *not* repeated in the original file.
−d	Write one copy of just the repeated lines.
−c	Supersedes −u and −d and generates an output report in default style but with each line preceded by a count of the number of times it occurred.
−*n*	The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.
+*n*	The first *n* characters are ignored. Fields are skipped before characters.

8.2. Modifying Files

The following commands can be used to modify files: `colrm`, `compact`, `compress`, `fold`, `pack`, `sort`, `split`, `tr`, and `tsort`. All of these commands are described in the *SunOS Reference Manual*.

8.3. Printing Files

To print files on paper, use the `lpr` command. To examine the print queue, use the `lpq` command. To remove jobs from the print queue, use the `lprm` command. All three of these commands are described in the *SunOS Reference Manual*.

Index