A/UX® Text Editing Tools

031-0124

Apple Computer, Inc.
20525 Mariani Ave.
Cupertino, California 95014
(408) 996-1010

Simultaneously published in the United States and Canada.

031-0124

031-0124

A/UX Text Editing Tools

Contents

Preface

Conventions Used in This Manual

Throughout the A/UX manuals, words that must be typed exactly as shown or that would actually appear on the screen are in `Courier` type. Words that you must replace with actual values appear in *italics* (for example, *user-name* might have an actual value of `joe`). Key names appear in CAPS (for example, RETURN). Special terms are in **bold** type when they are introduced; many of these terms are also defined in the glossary in the *A/UX System Overview*.

Syntax notation

All A/UX manuals use the following conventions to represent command syntax. A typical A/UX command has the form

`command` [*flag-option*] [*argument*]...

where:

`command`	Command name (the name of an executable file).
flag-option	One or more flag options. Historically, flag options have the form -[*opt*...] where *opt* is a letter representing an option. The form of flag options varies from program to program. Note that with respect to flag options, the notation `[-a][-b][-c]` means you can select one or more letters from the list enclosed in brackets. If you select more than one letter you use only one hyphen, for example, `-ab`.
argument	Represents an argument to the command, in this context usually a filename or symbols representing one or more filenames.
`[]`	Surround an optional item.

...	Follows an argument that may be repeated any number of times.
`Courier` type	anywhere in the syntax diagram indicates that characters must be typed literally as shown.
italics	for an argument name indicates that a value must be supplied for that argument.

Other conventions used in this manual are:

<CR>	indicates that the RETURN key must be pressed.
^*x*	An abbreviation for CONTROL-*x*, where *x* may be any key.
cmd(*sect*)	A cross-reference to an A/UX reference manual. *cmd* is the name of a command, program, or other facility, and *sect* is the section number where the entry resides. For example, `cat`(1).

Chapter 1
A/UX Text Editors: An Overview

Contents

Chapter 1

A/UX Text Editors: An Overview

1. What a text editor does

A **text editor** is a program designed to accept text you enter at the keyboard, store it, and allow you to modify it. There are two types of editing programs, ``interactive'' and ``stream.'' An **interactive editor** allows you to enter text and text-editing commands while you are viewing the text. A **stream editor** allows you a single pass over a document. Stream editing commands are usually kept in a file and you recall them from there, instead of having to enter them from the keyboard. Using the stream editor makes the editing process much faster, but you don't see your changes until the editor has finished with the entire document.

1.1 Using an interactive editor

To start using an A/UX text editor program, type the name of the editor program and a filename. If the file you name already exists, the editor opens that file in the editing buffer. Otherwise, it opens an empty file.

The **editing buffer** is a temporary work space, similar to a blank sheet of paper. When you create a file, you insert text into the buffer. When you modify a file, you make the changes to a copy of the file in the buffer.

1.1.1 Giving editor commands

When you begin an editing session, everything you type is interpreted as a command. Each A/UX editor has its own set of commands, but certain commands may be the same in all of them.

A/UX editor commands are usually single characters that stand for a function. These characters are mnemonic in almost every case; for example, d is the command for delete; w is the command for write, and so forth.

1.1.2 Entering text

When you open a file, the editor copies the file into the editing buffer.

You can modify the text in the buffer, insert new text, delete text, move blocks of text to new locations, make substitutions on a word every time it occurs, and so on. Remember that everything you do to the buffer contents is temporary until you write the contents back to the file. All three interactive A/UX text editors use the w command to write buffer contents to a permanent file on your disk.

There are several commands for entering new text into the buffer. The most common are `i` (for insert) and `a` (for append). After you enter one of these commands, everything you type is stored as text in the buffer. Each of the A/UX editors has a command to end text insertion and return to the interpretation of your editing commands.

1.2 Using a stream editor

A stream editor typically does not expect input from the keyboard in the same way as an interactive editor. Instead, a stream editor must be told where to look for its instructions. These are usually provided by storing them in a file (called an editing **script**), or, if they are few and simple, by giving them as flag options on the command line. The editor then performs the specified actions on the input files and writes the result to the standard output.

2. A/UX has four text editors

Your A/UX system has three interactive text editors: `ed`, `vi`, and `ex`. It also has a stream editor, `sed`. These are described briefly in the sections that follow. Each of these editors has a separate instruction guide and reference section in this manual. *Getting Started With A/UX* contains a tutorial for using `vi`.

2.1 `ed`: a line-oriented text editor

The `ed` program provides a single-line window into the text editing buffer. `ed` spends no time or system resources redrawing the screen, and can be an efficient way to enter text when you are working at 1200 baud or lower. It has a limited number of commands, but these can be combined to perform most of the tasks you need most frequently. You can also use `ed` commands in a shell program, since it does not operate on a full screen of text.

2.2 vi: a screen-oriented text editor

The vi program provides a full-screen window into the buffer. Every change you make to the buffer contents is immediately displayed, and the screen image is updated. This is much more convenient than a single-line image that does not display your changes. However, it requires more system overhead; this may affect you if you are working at 1200 baud or lower, or if your system is very busy.

vi is derived from the ex editor program, and most ex capabilities are directly accessible while you are using vi. In addition to the ex commands, vi has many motion commands that allow you to move around the file by character, word, line, sentence, paragraph, or section or move to a particular character string. Most of vi's motion commands can be combined with function commands (such as d for delete) to define the scope of an operation.

2.3 ex: a line-oriented text editor

The ex program provides a single-line window into the text editing buffer and has the advantages of reduced system overhead and accessibility from a shell program.

ex is a powerful editing tool for substitutions and global commands. ex can search for a pattern and perform substitutions on any string that matches that pattern. This greatly increases the power and flexibility of substitution commands.

ex has options that define the ''editing environment'' for the ex and vi editors (such as the margin for word wraparound on your screen, automatic indent following a line that starts with a tab character, making line numbers visible, a special environment for editing programs, and so on). ex also has macro facilities for ''mapping'' a key to perform a complex editing sequence or abbreviate a long string to a short one.

Remember that all of these ex capabilities are accessible while you are using vi and can also be accessed from shell programs using the ex editor alone.

2.4 sed: a stream editor

sed is a stream editor: it copies the input file(s) to the standard output, performing various user-specified editing tasks (such as substituting or deleting words) on the file as it ''flows'' by. These tasks may be specified on the command line or, more commonly, stored in a file for repeated use.

Because of its batch nature, `sed` is extremely useful for building ``filters'' to edit or modify text without user supervision. Thus, `sed` may be run in the background, allowing the user to perform other tasks while the editing takes place. `sed` is also useful for editing very large files, since no buffer is created.

The changes specified in the `sed` script (or on the command line) affect only a copy of the file, not the original file itself. The output of the `sed` command is directed to the standard output; this is usually your terminal screen, but you may redirect the output into a file. Or, you may redirect the output of a `sed` command into a pipeline to allow further filtering by other A/UX utilities.

Chapter 2
Using ed

Contents

Chapter 2

Using ed

ed is an interactive line-oriented text editor that uses your instructions to create and modify text files. A **line-oriented text editor** moves through your file one line at a time and allows you to modify that line or to change another line or range of lines (indicated by line number). The red editor is a restricted version of ed. It is identical to ed except you can only edit files in the current directory and you cannot access shell commands.

This chapter summarizes the capabilities of the text editor ed, including the following:

- printing, appending, changing, deleting, moving, and inserting text
- reading from, and writing to, files
- searching for text
- making substitutions
- making global changes throughout a file
- using special characters for easier editing

This chapter assumes that you know how to log in to an A/UX system and understand what a file is. Examples illustrate the techniques discussed in the text.

Note: Except for the command you use to invoke the editor program, all commands discussed in this chapter are commands to ed. Do not confuse them with A/UX shell commands.

A summary of ed commands appears at the end of the chapter.

1. Getting started

To start `ed`, type

```
ed
```

(followed by RETURN).

> *Note:* Unless explicitly instructed otherwise, conclude all `ed` commands with a RETURN.

You can also invoke `ed` with a filename as an argument:

`ed` *filename*

where *filename* may or may not already exist.

If a file by that name does not exist, you see the message

`?`*filename*

If a file by that name does exist, `ed` displays the character count on the screen.

1.1 Displaying a prompt

You can use the `P` command to display a prompt on your screen. Type

```
P
```

The following appears on the left side of your screen:

```
*
```

You type `ed` commands next to the asterisk (`*`) in the same way that you type shell commands next to the A/UX system prompt. (See *Getting Started With A/UX* for a discussion of the shell prompt.) To turn off the prompt, type the `P` command again.

1.2 Error messages

If `ed` doesn't understand something you type, it prints a question mark (`?`) on the screen.

For assistance in interpreting this error message, type

```
H
```

The H (help) command explains the current ? and all subsequent ones. Typing the H command again turns off this feature.

Alternatively, you can use the h command. This form of the help command explains only the current ?.

1.3 Creating text

When you start ed, you ''open'' the editing buffer. The buffer corresponds to an empty file. It is a temporary work space, similar to a blank piece of paper. When you create a file, you must insert text into the buffer or read it in from another file and then save the new or modified data.

When you give the command

```
ed filename
```

where *filename* is an existing file, ed makes a copy of this file and places it in the editing buffer. Any modifications or additions you make to this file are made on the copy, not on the original file.

The following example begins with inserting text in an empty buffer (editing existing files is discussed later).

To begin creating text, type

```
a
```

on a line by itself, and press RETURN. (The a command means ''append'' or ''add'' text lines to the buffer as they are typed in.)

Type the following text:

```
A journey of a
thousand miles
begins with a
single step.
.
```

As shown in the last line of this example, appending is stopped by typing a period character (.) followed by RETURN. The period character must be the first and only character on the line. This tells ed that you have finished adding text and are ready to give a command. Even experienced users sometimes forget to type the period character when they have finished adding text. If ed seems to be ignoring your commands, type a period, and then press RETURN. You may find that some command lines in your text have to be removed.

After you finish appending, the buffer contains these four lines:

```
A journey of a
thousand miles
begins with a
single step.
```

To add more text, type

```
a
```

(RETURN), and continue typing.

1.4 Saving text

After you have added text to the buffer, you will want to save it. The w (write) command writes the contents of the buffer into a file. For example, if you type

```
w text
```

the buffer's contents are copied into a file named `text`.

If you named your file when you began your editing session, or if you are editing an existing file, you don't have to repeat that filename when you write the file. `ed` remembers the original filename you designated and automatically reuses it. For example, an editing session might look like this:

```
ed text
(editing session)
.
w
```

The file you edited is saved in a file named `text` when you type `w`.

You can also use the `w` command to save part of a file. The `w` command writes the lines you specify from the buffer to the permanent file. If no lines are specified, the `w` command writes the entire file. For example, the command

```
1,10w
```

saves the first ten lines of your file.

In another example, if you are editing your file `text`, and you give the command

```
1,10 w another.file
```

`ed` writes the first ten lines of your file `text` to the file `another.file`.

Note that when you assign a name to a file from within `ed` you must make sure that you do not have an existing file by that name. The write command replaces that file with the current buffer's contents *without* giving you a warning.

After writing the file, `ed` responds as follows:

```
57
```

This represents the number of characters (including blank spaces and end-of-line characters) that were written into the file.

> *Note:* It's a good idea to write your text to a file every 10 or 15 minutes. If the system crashes or you make a mistake, you may lose the text in the buffer, but any text in a file should be safe.

1.5 Leaving ed

To leave `ed` after saving your text with the w (write) command, quit your file by typing

```
q
```

(followed by RETURN). For example, in the editing session described above, the following appears on the screen:

```
ed                    (start the editor program)
a                     (append )
A journey of a        (text)
thousand miles        (text)
begins with a         (text)
single step.          (text)
.                     (end append)
w text                (write to a file named text)
57                    (character-count system response)
q                     (quit)
```

When you leave ed, the buffer is destroyed, and the system responds with its usual shell prompt character.

If you try to quit the editor without writing the buffer contents to a file, ed prints

```
?
```

on your screen.

If you don't want to save the changes to your file, typing q a second time (followed by RETURN) gets you out of ed and back to the shell without saving the changes you made since the last w command. If you want to save the changes to your file, type w and press RETURN.

2. Using ed to modify a file

After you have created and saved a file, you may want to edit it. There are two ways to do this.

To edit a file from the shell, type

ed *filename*

(followed by RETURN). This retrieves a file you previously saved and places it in the buffer.

Another method of editing a file from within ed is by typing

e *filename*

When you use the e command to edit a file, ed replaces the contents of the buffer with the new file. If you were already working on a file in the buffer and you haven't written it yet, the e command destroys it without warning you.

If you forget the name of the file you have in the buffer, you can find out using the `f` (file) command. From within the editor, type

```
f
```

and the name of the file appears on the screen.

2.1 Printing buffer contents: using line numbers

To display all or part of the buffer on your screen, use the `p` (print) command. You must specify the line numbers where you want printing to begin and end. Separate these numbers with a comma in this format:

line1`,`*line2*`p`

Through this chapter, such line addressing is represented with the following:

line1`,`*line2command*

where *command* is `p` in this case. *line1*`,`*line2* indicates a range of addresses from *line1* to *line2*.

For example, to print the first ten lines of the buffer (lines 1 through 10), type:

```
1,10p
```

You can also tell `ed` to display the line numbers of the lines you specify with the `p` command. For example,

```
2,4pn
```

prints the following:

```
2                    text of line 2
3                    text of line 3
4                    text of line 4
```

Suppose you want to print *all* the lines in the buffer. If you know the exact number of lines in the buffer, such as 30, you could type `1,30p`. However, if you don't know how many lines there are in your file, use the dollar sign (`$`). (The dollar sign refers to the last line of the file; see the section ''Special Characters'' in this chapter.) To print all the lines in the buffer, type

```
1,$p
```

To stop printing, press the *interrupt* key (usually CONTROL-c). ed responds with

```
?
```

and waits for the next command.

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

```
$p
```

You can print any single line by typing the line number. For example, typing

```
1
```

prints

```
A journey of a
```

which is the first line of the buffer.

In ed, the current line is the most recent line processed (in this case, the line last printed). If you type p again, ed prints line 1 again. The period character (or ''dot'') always refers to the current line. It is a line number in the same way that $ is. You can use dot in several ways—one possibility is to enter

```
.,$p
```

This prints everything from the current line to the last line of the buffer. In the example text file, these are lines 1 through 4.

Some commands move the current line to a new place in the file (that is, they change the value of dot); others do not. The p command resets dot to the number of the last line printed. For example

```
.,$p
```

sets dot to the last line in the buffer (line 4).

Dot is most useful in combinations such as

```
.+1                (this is equivalent to .+1p)
```

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

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

which means "print the line before the current line." This allows you to move backward through the buffer. Another useful example is

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

which prints the previous three lines.

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

```
    .=
```

This will print the line number of the current line. Pressing RETURN once prints the next line. It is equivalent to

```
    .+1p
```

To summarize, you can precede `p` by zero, one, or two line numbers. If you don't specify a line number, `p` prints the current line (the line that dot refers to). If you specify one line number with or without the letter `p`, `ed` prints that line and makes it the current line. If you specify two line numbers separated by a comma and followed by `p`, `ed` prints everything from the first number to the last number, and sets dot to the last line printed. (The first number must be smaller than the second number—`ed` won't print backward.)

Typing the caret (^) or the minus sign (−) moves the current line back one line. These characters can be used in multiples; typing ^^^ or −−− moves the current line back three lines. The minus (−) and caret (^) are the same as `-1p`.

You can use line numbers with most `ed` commands, as you will see in the sections that follow.

2.2 Reading text into the buffer

If you want to add an existing file to the buffer without overwriting what is already there, use the `r` (read) command. The command

```
    r new.file
```

adds the contents of the file `new.file` to the end of the file already in the buffer. If you type

```
e text
57              (system response)
r text
57              (system response)
```

the buffer now contains two copies of the same file:

```
A journey of a
thousand miles
begins with a
single step.
A journey of a
thousand miles
begins with a
single step.
```

Like the `w` and `e` commands, `r` prints the number of characters that it read into the buffer.

If you precede the `r` command with a line number or a dot (`.`), it reads a file and puts it after the specified place in the current buffer.

`.r` *filename*

reads the contents of *filename* into the buffer immediately after the current line. (In this context, dot is equal to the current line. This is different from the period character on a line by itself, which means that the text insertion is over.)

`3r` *filename*

reads the contents of *filename* into the buffer following line number 3.

The file in the buffer is not destroyed—it continues after the last line of the file you read in. For example, using the original `text` file

```
ed text
57                              (system response)
1                               (go to line 1)
A journey of a                  (system response)
.r text
57                              (system response)
w
114                             (system response)
q
```

places this in your file:

```
A journey of a
A journey of a
thousand miles
begins with a
single step.
thousand miles
begins with a
single step.
```

2.3 Deleting text

The d (delete) command removes lines of text from the buffer. The d command uses the same format as the p command

line1, *line2* d

For example, the command

```
4,$d
```

deletes everything from line 4 to the end of the buffer. In the preceding example, this deletion leaves us with three lines. We can check these lines by typing

```
1,$p
```

The last line, $, is now line 3. If you delete the last line (as in the preceding example), dot is set to $.

You can use the d (delete) command and the p (print) command together. For example, typing

```
dp
```

deletes the current line, prints the next line, and sets dot to the line printed.

2.4 Inserting text

The `i` (insert) command inserts one or more lines into the buffer. It is similar to the `a` command except that it places the text *before* rather than *after* the current or specified line—for example, typing

```
2i
one or more lines of text
.
```

inserts the text *before* the second line. If you don't specify a line number, the text is inserted before the current line. Dot is set to the last line inserted.

Experiment with the `i` and `a` commands to see how they operate. Verify that

```
line-speca
text
.
```

appends *after* the given line, while

```
line-speci
text
.
```

inserts *before* it, where *line-spec* indicates a single line number or a scanning command (such as a context search or regular expression) resulting in zero or more lines. If a line number isn't specified, the current line is assumed.

2.5 Changing text

The `c` (change) command changes the current line, replacing it with one or more lines. For example, to replace everything between the current line and the last line, type

```
.+1,$c
```
one or more lines of text

```
.
```

The text you type between the `c` command and the `.` command will overwrite the original text from the `.+1` line to the last line. This command is useful when you want to replace one line or several lines.

If you specify only one line, only that line is replaced. (You can type as many replacement lines as you like.) Notice that you end your changes by typing a period (`.`) at the beginning of a line—this is the same way you stopped adding text with the `a` command.

The `c` command can also be thought of as a combination of the `d` command followed by the `i` command. Experiment to verify that

line1`,`*line2*`d`
`i`
text
`.`

is the same as

line1`,`*line2*`c`
text
`.`

If you don't specify a line number, `c` replaces the current line. When you finish making changes, dot is set to the last line you inserted.

2.6 Text substitution

One of the most important `ed` commands is the `s` (substitute) command.

This command changes words or characters and can be used to correct spelling mistakes and typing errors.

Suppose that line 1 is

```
A journy of a
```

You can change `journy` to `journey` by typing

```
1s/ny/ney/
```

This says: in line 1, change `ny` to `ney`. Since `ed` doesn't print the change automatically, type

```
p
```

to make sure the substitution worked. You should see

```
A journey of a
```

When you include the `p` command on the same line as the substitute command

```
s/journy/journey/p
```

`ed` prints the line that just changed.

The general format of the substitute command is

line1 , *line2* `s` / *change this* / *to this* /

The characters between the first and second slashes (*change this*) are replaced by the characters between the second and third slashes (*to this*). This substitution takes place on *all* lines between *line1* and *line2*. However, only the *first* occurrence on *each* line is changed. To change *every* occurrence, on *each* line, add `g` (global) (see ''Global Commands'') to the `s` command, like this:

line1 , *line2* `s` / *something* / *something-else* / `g`

The rules for line numbers are the same as those you learned for the `p` (print) command. However, if the `s` command can't find the characters you asked it to change, the cursor stays in the current position. `ed` tells you when this has happened by printing `?` on the screen.

As an example of a substitution, you could type

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

to correct the *first* instance of `speling` on each line. (This is useful for people who make the same mistake consistently.)

If you don't specify a line number, `s` assumes you want to make the substitution on the current line. For example, you could type

`s` / *something* / *something-else* / `p`

This corrects a mistake on the current line and then prints it to verify that the substitution worked.

You may have noticed that the s command resets the current line. You can also type

```
s/something//
```

This replaces *something* with nothing—in other words, it removes *something*. This is useful for deleting extra words in a line or removing extra letters from words.

For example,

```
Thisxx is an example of substitution
```

can be corrected by typing

```
s/xx//
```

The line now reads

```
This is an example of substitution
```

The // (two adjacent slashes) mean "no characters," *not* a blank.

Experiment with the s command. For example, type

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

This produces the following:

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

Remember that the s command changes only the first occurrence. You can change all occurrences on a line by adding g.

Try using characters (except blanks and tabs) other than slashes to set off the two sets of characters in the s command. For example, try typing

```
s'the'other'p
```

This works exactly the same as using a slash.

However, strange results are produced by using the backslash (\) character (see the section "Special Characters" in this chapter).

2.7 Global commands

The g (global) command performs an operation on all lines that match a specified string or regular expression. See Chapter 5, "Using ex" for information on regular expressions; in this chapter we use the word "string" to mean a string of characters or a regular expression. For example,

```
g/speling/p
```

prints all lines that contain speling. The command

```
g/speling/s//spelling/gp
```

replaces speling with spelling each time it occurs (even if it occurs more than once in a line), then prints each corrected line.

Compare this to

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

This prints only the last line substituted.

You can use several commands at a time with g. Just remember to end every line but the last with a backslash (\). For example,

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

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

The G (interactive global) command finds a line that matches a specified string, prints the line, and waits to accept a command. After executing the command, it searches for the next line that matches the specified string, and so on. For example,

```
G/speling/
```

prints the first line that contains the string "speling." If you wish to change the string at that point, you can enter the command

```
s/speling/spelling/p
```

which replaces `speling` with `spelling` and prints the corrected line. After printing the corrected line, `ed` searches for the next instance of "`speling`." If found, it prints the line that contains this string, and waits for you to enter a command. The command you enter does not have to be the same command you entered last time; for example, if `ed` finds another instance of `speling`, you could enter the command

```
s/speling/misspelling/p
```

or any single `ed` command other than the `a`, `c`, `i`, `g`, `G`, `v`, or `V` commands.

The `v` command is the same as `g` except that it executes the commands on lines that *don't* match the string or regular expression. For example,

```
v/ /d
```

deletes every line that does not contain a blank. Similarly, the `V` command is the same as `G`, but finds and prints lines that don't match the specified string or regular expression.

2.8 Context searching

When you master the substitute command, you may want to try another important feature of `ed`—context searching. Context searching looks for a string of characters and, when it finds it, makes that line the current line.

Suppose you have these three lines in your buffer:

```
Little Miss Muffet
sat on a tuffet
eating her curds and way.
```

If you want to locate the misspelled word `way`, you could type `3`. However, if the buffer contained several hundred lines and you had been deleting and rearranging lines, you might have a difficult time locating this line. Context searching lets you find a line by specifying some context (unique text) in it.

To search for a line that contains a particular string of characters, type

/string of characters

For example,

```
/way
```

locates the next occurrence of `way`. It also makes that line the current line and prints it for verification.

"Next occurrence" means `ed` starts looking for the string at the line following the current line (`.+1`) and searches to the end of the buffer. Then it searches from line 1 to the line it started searching at (dot). That is, the search *wraps around* from `$` to `1`. It scans all the lines in the buffer until it either finds the desired string or gets back to dot again. If `ed` can't find the characters, it types the error message

```
?
```

Otherwise, it prints the line it found.

You can search for the desired line *and* make a substitution to it in the same command, like this:

```
/curds/s/way/whey/p
```

This tells `ed` to search for the line with the word `curds`, substitute `whey` for `way`, and then print the new line. When it has finished, `ed` prints this:

```
eating her curds and whey.
```

You can repeat a context search. For example,

/*string*/

finds the next occurrence of *string*. If this is not the line you want, you can search for the next occurrence by typing

`//` or `/`

This stands for "the previous context search expression" and differs from the use of `//` as a null argument in the `s` command.

This abbreviation can also be used as the first string of the `s` command. For example,

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

finds the next occurrence of `string1` and replaces it with `string2`.

Similarly,

```
?? or ?
```

scans backward for the previous expression.

You can use context searches instead of line numbers to find a desired line or to specify a range of lines to be affected by some other command, such as `s`.

For example, suppose the buffer contains these four familiar lines:

```
A journey of a
thousand miles
begins with a
single step.
```

The following context search expressions all refer to the same line (line 2):

```
/journey/+1
/thousand/
/step/-2
```

To make a change in line 2, you can type

```
/journey/+1s/thousand/hundred/
```

or

```
/thousand/s/thousand/hundred/
```

or

```
/step/-2s/thousand/hundred/
```

You could print all four lines by typing either

```
/journey/,/single/p
```

or

```
/journey/,/journey/+3p
```

The first of these might be better if you don't know how many lines there are. A context search expression is the *same* as a line number, so it can be used wherever you would use a line number.

2.9 Moving text

The `m` (move) command moves lines from one place to another. For example, to move the first four lines of the buffer to the end, type

```
1,4m$
```

The general case is

line1`,`*line2*`m` *lineno*

The text is moved after the specified line number (*lineno*). You can use context searches instead of line numbers. For example, if you have the following text in your buffer,

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

you could reverse the two paragraphs by typing

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

The `-1` was used because the text is moved *after* the line specified. Dot is set to the last line moved.

3. Special characters

You may have noticed that some characters (such as `.`, `*`, `$`) change the meaning of context searches and the `s` command. This is because these characters have special meanings for `ed`.

The following is a complete list of these special characters:

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

These are described in the sections that follow.

3.1 Period

In a context search or the first string of the substitute command, the period (`.`) signifies *any* character.

Although this is the same character as "dot," its meaning is different in this context. To avoid confusion, we call it *dot* when it means "current line" or "line most recently changed" and *period* when it means "any character."

```
/x.y/
```

means

/x*any-character*y/

This command will find all instances of x followed by any character followed by y, including the following:

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

3.2 Caret

The caret (^) signifies the beginning of a line. For example,

/^*string*/

finds *string* only if it is at the beginning of a line. That is, it will find

```
string
```

but not

```
the string
```

3.3 Dollar sign

The dollar sign ($) is the opposite of the caret; it means the end of a line.

The expression

```
/string$/
```

finds `string` only at the end of a line.

```
/^string$/
```

finds a line containing only `string` and

```
/^.$/
```

finds a line containing one character.

3.4 Asterisk

The asterisk (`*`) is the repetition character. For example, `a*` means ``zero or more a's.'' `.*` means ``any character repeated zero or more times.''

For example,

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

changes an entire line to `stuff`, and

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

deletes all the characters in the specified line up to, and including, the last comma. Note that `.*` finds the longest possible match, so this example matches the last comma rather than the first.

3.5 Brackets

The left bracket (`[`) is used with the right bracket (`]`) to enclose ``character classes.'' For example,

```
/[0123456789]/
```

searches for any single digit. This can be abbreviated as

```
[0-9]
```

Brackets can also be used to contain a character class that represents the alphabet; for example,

```
[A-Z]
```

searches for any uppercase character, and

```
[a-z]
```

searches for any lowercase character.

3.6 Ampersand

The ampersand (`&`) means ``whatever was matched on the left-hand side.'' (The ampersand only has this meaning on the right-hand part of a substitute command.)

Suppose the current line contains

```
Now is the time
```

and you want to put parentheses around it. You can accomplish this using the command

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

This says "match the whole line (`.*`) and replace it by itself (`&`) surrounded by parentheses." The ampersand can be used several times in a line. Using the preceding sample text,

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

produces

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

You don't have to match the whole line. If the buffer contains

```
the end of the world
```

you could type

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

to produce

```
the end of the world is at hand
```

The sequence `/world/` found the desired line; the sequence `//` found the same word in the line; and the `&` saved you from typing `world` again.

The `&` is a special character only in the replacement text of a substitute command.

3.7 Backslash

If you have to use one of the special characters listed above without its special meaning in a substitute command, precede it with a backslash (\). For example,

```
s/\.H//
```

replaces the first occurrence of a `.H` with nothing (`//`) on the current line (in other words, it deletes it). If the period (`.`) were not preceded by the \, the result would have been that the first instance of `H` preceded by *any* other character would have been deleted on the current line.

4. Command summary

In the following summary, *line-spec* indicates a single line number or a scanning command (such as a context search or regular expression) resulting in zero or more lines; *line1*, *line2* indicates a range of addresses from *line1* to *line2*. If you don't specify an address, the current line is the default (unless otherwise noted). The <CR> symbol indicates that the RETURN key must be pressed. Portions of a command enclosed in brackets ([]) are optional.

Command	Description
[*line-spec*]a<CR>[*text*]<CR>.	Append text after the current line or after the line number specified. To stop adding text, type a period (.) at the beginning of a line, and press RETURN. Dot is set to the last line appended.
[*line-spec*]c<CR>[*text*]<CR>.	Change the specified lines to the new text which follows. To stop replacing text, type a period (.) at the beginning of a line, and press RETURN. If you don't specify a line, the current line is replaced. Dot is set to the last line changed.
[*line1*, *line2*]d	Delete the specified lines. If you don't specify a line, the current line is deleted. Dot is set to the line after the last deleted line. If you delete the last line in the buffer, dot is set to the new last line.
e *file*	Edit a new file from within ed. The previous contents of the buffer are destroyed, so save your work before you edit a new file with e.

Command	Description
`f` [*file*]	Print the current filename. This is the file `ed` assumes you mean if you don't specify a file. To change the current filename, type `f` *file*.
[*line1*, *line2*]`g`/*string*/*command*	Execute commands globally, on the entire file (by default). `g/x/`*command* executes *command* on lines containing the string `x`.
[*line1*, *line2*]`G`/*string*[/]	Interactive global command. `ed` first marks every line that matches the given regular expression or string. Then, for every such line, that line is printed, dot is changed to that line, and any *one* command (other than one of the `a`, `c`, `i`, `g`, `G`, `v`, and `V` commands) may be input and is executed. After the execution of that command, the next marked line is printed, and so on; a RETURN acts as a null command (no action is performed); an `&` causes the re-execution of the most recent command executed within the current invocation of `G`. Note that the commands input as part of the execution of the `G` command may address and affect *any* lines in the buffer. The `G` command can be terminated by an *interrupt*. A command that causes an error terminates the `G` command.
`h`	The `h` (help) command gives a short error message that explains the reason for the most recent `?`.

Command	Description
`H`	The `H` (Help) command prints error messages for all subsequent `?` diagnostics. This command toggles error message printing on and off.
[*line-spec*]`i`<CR>[*text*]<CR>`.`	Insert text before the specified line or the current line. To stop inserting text, type a period (`.`) at the beginning of a line, and press RETURN. Dot is set to the last line inserted.
[*line1*, *line2*]`j`	Join contiguous lines by removing appropriate newline characters.
[*line-spec*]`k`*x*	Mark addressed line with name *x*, which must be a lowercase letter. The address *x* then refers to this line; dot is unchanged.
[*line1*, *line2*]`m`*lineno*	Move the text originating between *line1* and *line2* to follow *lineno*. Dot is set to the last line moved.
[*line1*, *line2*]`n`	For the current line or for each line in the range specified by "*line1*, *line2*," print the line number, followed by a tab, followed by the text of the line(s).
[*line1*, *line2*]`p`	Print the specified lines. If you don't specify any line number, `p` prints the current line. Pressing RETURN prints the next line.
`P`	Turns prompting on and off. The `P` command alternately turns this mode on and off; initially it is off.

Command	Description
q	Quit ed. No automatic write of a file is done. If changes have been made in the buffer since the last w command, ed responds with ?. If you don't want to save your changes, type q <CR> again.
Q	Quit without checking to see if changes have been made in the buffer since the last w command.
[*line-spec*]r *file*	Read a copy of *file* in at the specified location. If no line number is specified, it reads the file in at the end of the buffer. Dot is set to the last line read.
[*line1*, *line2*]s/*string1*/*string2*[/]	Substitute one string for another string at a specified location. 1,$s/*string1*/*string2*/g substitutes *string2* for every instance of *string1* in the file. The s command changes only the first occurrence of *string1* on a line. To change all occurrences, type g at the end of the command. Dot is set to the last line in which a substitution took place; if no substitution took place, dot is not changed.
[*line1*, *line2*]t*lineno*	Put a copy of the addressed lines after address *lineno* (which may be 0); dot is left at the last line copied.
u	Undo last command; nullifies the effect of the most recent command that modified anything in the buffer.

Command	Description
[*line1*, *line2*]v/*string*/*command*	Execute *command* only on lines *not* containing *string*. By default the v command operates on the entire file.
[*line1*, *line2*]V/*string*[/]	Interactive global command marks each line not containing *string* and then allows you to perform commands on each of these lines. By default the V command operates on the entire file.
[*line1*, *line2*]w *file*	Write the buffer into the specified file. Dot is not changed. By default the w command writes the entire file.
X	Request an encryption key string from the standard input. Subsequent e, r, and w commands encrypt and decrypt the text with this key by the algorithm of crypt(1). An explicitly empty key turns off encryption.
[.]=	"Dot equals" prints the current line number. = by itself prints the line number of the last line in the file.
!*shell-command*	Temporarily escape to the A/UX shell to execute the specified command. !*shell-command* executes *shell-command* in the shell and then returns you to the editor.

Command	Description
/*string*[/]	Search through the file for *string* and print the line containing it. The search starts at the line after the current line, reads to the end of the buffer, then wraps around to line 1 and searches to the original line. If *string* is located, dot is set to the line where the string is found.
?*string*[?]	Search backward through the file for *string* and print the line containing it. The search begins at the line before the current line, reads backward to the start of the file, then wraps around to the end of the file and searches backward to the original line. If *string* is located, dot is set to the line where the string is found.

Chapter 3
`ex` and `vi`:
A Text Editing System

Contents

Chapter 3

ex and vi:

A Text Editing System

1. What are ex and vi?

ex and vi are actually two aspects or **modes** of the same text-editing program. For convenience they are often referred to as two separate programs.

- vi is a screen editor. A screen editor works by displaying the contents of a file a full screen at a time. You type commands to add or change text anywhere on the screen, and the screen changes immediately to show the changes. Most of the time you need not know the line numbers of the lines you wish to work on. For our purposes, the terms ''vi'' and ''visual mode'' both refer to the vi screen editor.

- ex is a line editor. A line editor works by specifying a set of lines on which to operate (for example, add text after this line, or change these ten lines). You issue commands to add or change text in response to a command prompt, and you cannot always see the results of changes right away. In most cases, you'll need to know the line numbers of the lines you wish to modify or otherwise operate on. For our purposes, the terms ''ex'' and ''line mode'' (which may be accessed within vi) both refer to the ex line editor.

Both modes have commands you use to enter and edit text. Generally, the command to perform a given action in one mode is similar to the command you would use in the other mode (for example, d to delete text).

You can go back and forth between ex and vi. To go from ex to vi, you type

```
vi
```

at the ex colon prompt.

To use ex from within vi, press the ESCAPE key (if necessary) to enter vi command mode and then type the colon character (:). The ex command line appears at the bottom of the screen, ready to accept any ex command. The current screenful of text remains on the screen.

ex commands are invaluable in vi for global changes, searching, and other operations that involve more than one line in the file. vi is helpful in ex as well.

2. In this manual

The complete reference manual for this text-editing system consists of the following two chapters:

- Chapter 4, ''Using vi,'' discusses vi commands. Where an ex command is useful, it is mentioned and you are referred to ''Using ex.''
- Chapter 5, ''Using ex,'' covers ex commands. It includes a command summary in the back. Where an ex command involves or is related to a vi command, you are referred to ''Using vi.''

These two chapters assume that you've at least tried using vi and ex before. If you've never used them, please go through the vi tutorial in *Getting Started With A/UX*. After you've finished the tutorial, you'll be ready for these chapters.

Chapter 4
Using `vi`

Contents

Chapter 4

Using `vi`

1. `vi` basics

This chapter describes the visual editor, `vi`. This chapter assumes you've used `vi` at least once and are ready to learn more about what `vi` has to offer. If you've never used `vi`, please go through the `vi` tutorial in the *Getting Started With A/UX* manual. After you complete the tutorial, you'll be ready for this chapter.

`vi` uses the full screen as a window into the file you edit. When you make a change, `vi` immediately displays the change on your screen. The `view` command is similar to `vi` but protects you from making unintended changes by setting read-only permission on the file. The `vedit` command is identical to `vi` except that an "`INPUT MODE`" message displays when you are entering text, and `vedit` reports the number of changes you make with global substitutions (when the number of changes is greater than one). The `vedit` command is intended to be helpful to beginning users.

1.1 Starting `vi`

Usually you start `vi` with the following command:

`vi` *filename* ...

where *filename* is the name of the file(s) to edit.

1.1.1 `vi` command-line syntax

The command to invoke `vi` is

`vi` [+*command*] [`-l`] [`-r` [*filename*]] [`-R`] [`-t` [*tag*]] [`-w`*n*] [`-x`] [*filename* ...]

You can also use the `view` and `vedit` commands with the same flag options.

The options are as follows:

+*command*	Move to the line specified by *command* where *command* is either a regular expression (see "Regular Expressions and Searching") or a line number (for example, `+100` starts editing at line 100). If you omit *command*, `vi` moves the cursor to the last line of the first file.
`-l`	Set the `showmatch` and `lisp` options for editing LISP programs. These are described under "Setting Options."
`-r` [*filename*]	Recover a file after an editor or system crash; if you don't specify *file*, it lists the saved files.
`-R`	Set the `readonly` option, making it impossible to write the file with the `write` command.
`-t` [*tag*]	Start editing the file at *tag* (usually a spot marked with the `ctags` program). Equivalent to an initial `tag` command. This is described in Chapter 5, "Using `ex`."
`-w`*n*	Set the window size to ***n*** lines.
`-x`	Prompt for a *key* to encrypt and decrypt the file (see `crypt`(1) in the *A/UX Command Reference*). The file should already be encrypted using the same key.
filename	The file(s) to edit.

1.1.2 `vi` Initialization

When you start `vi`, it sets up your editing environment with the following steps:

- reads the `TERM` variable to find out what terminal you're using
- sets any options you've specified in the `.exrc` file in the current directory or your home directory
- sets any options you've specified in the `EXINIT` variable (usually set in the `.profile` (or `.login`) file in your home directory)

You can set the same options with either the `.profile`, `.login`, or `.exrc` files. The options are described in "Setting Options" later in this chapter.

1.2 Opening a file

You can display a file for read-only viewing or to edit and make changes. This is the difference between **viewing** a file and editing it.

1.2.1 Read-only viewing

If you want to look at a file and protect the file from unintended changes, start the viewing program from the shell by typing

`view` *filename*

instead of `vi` *filename*. `view` protects the file from accidental changes. When you enter a file using `view`, you can use all of `vi`'s commands, but you can only make changes to the file by typing the colon character (`:`) to move to line mode and using the command

```
:w!
```

You can then exit the file by typing the colon character (`:`) to move to `ex` line mode and typing

```
q
```

If you exit `view` using the `vi` command

```
ZZ
```

You will exit the file without making any permanent changes. If you try to exit `view` using commands that write changes to a file before exiting `vi` (for example, `:wq`), you'll get an an error message.

1.2.2 Opening a file for editing

To create and "open" a new file (or open an existing file) in `vi`, type

`vi` *filename*

where *filename* is the name of the file you're creating (or opening). For example, to open the file `jumblies` from the `vi` tutorial in *Getting Started With A/UX*, type

```
vi jumblies
```

When you use `vi` to create a new file, `vi` opens some temporary storage space that is referred to as "the buffer."

When you edit an existing file, `vi` places a copy of that file in the buffer. Changes you make to the text in the buffer (for example, to a copy of the `jumblies` file mentioned above) are made only to this temporary copy. `vi` does not change the actual contents of the file until you save your changes (see "Saving Text and Exiting").

> *Note:* You should periodically write your changes to the file to prevent losing material if the system crashes or is interrupted.

After you've opened an existing file the text is displayed on your screen.

1.3 `vi` modes

`vi` has a number of modes.

- When you first open a file, you are in **command mode.** `vi` assumes anything you type is a command and tries to execute it. For example, if, after you've created and opened a new, empty file you type the letter `j`, which happens to be a movement command (discussed later in this chapter), `vi` tries to move accordingly. But because there is no text in the file, there is no place to move. `vi` signals that it cannot comply with the command.
- To insert text, you must enter insert mode. In **insert mode,** `vi` assumes anything you type is text (rather than commands). To enter insert mode, type `i`. Anything you type after that appears on the screen. `vi` places what you type in the buffer. It won't be written to the file until you return to command mode and save the file (see "Saving Text and Exiting"). Other `vi` commands for inserting text include `a` (append), `o` and `O` (open line). There are also several commands, such as `c` (change) and `s` (substitute), that insert text. For a complete list of `vi`'s insertion commands, see "Inserting Text" later in this chapter. You always leave insert mode by pressing the ESCAPE key.
- You can use the `ex` editor commands by entering line mode. See "Switching to Line Mode."

1.4 Switching to line mode

When you invoke `vi` you can switch to line mode by typing the colon (`:`) in `vi` command mode. This invokes line mode and places a colon prompt on the bottom line of your screen. You can enter `ex` commands at this prompt. After a single `ex` command has executed, you will return to `vi`.

You can also switch to line mode for a series of `ex` commands (or a multiple-line `ex` command) by typing

```
Q
```

in `vi` command mode. To return to `vi` when you have entered line mode this way, type

```
vi
```

at the colon prompt on the bottom line of your screen.

`ex` commands are invaluable in `vi` for global changes, searching, and other operations that involve more than one line in the file.

> *Note:* You can also switch to the `ex` command line to execute `ex` search and `global` commands by typing the slash character (`/`) or the question mark (`?`).

For complete information on using `ex` commands, refer to Chapter 5, "Using `ex`."

1.5 Special keys

ESCAPE

The ESCAPE key ends all text insertion in `vi` and returns you to command mode. Try pressing this key a few times. On most terminals a bell sounds (on some terminals the screen silently flashes instead), indicating that you are in command mode.

RETURN

The RETURN (carriage return) key terminates all commands given on the `ex` command line. See "Switching to Line Mode."

> *Note:* You do not need to press RETURN after commands that are not given on the `ex` command line.

The *interrupt* key

The *interrupt* key (set to CONTROL-c in the A/UX standard distribution) sends an interrupt signal to the editor. It gives you a forceful way of stopping `vi` from executing a command it has already started.

`vi` occasionally shows your commands on the last line of the screen. If the cursor is on the first position of this last line, `vi` is working on something (such as finding a new position in the file after a search or reformatting the buffer). When this happens, you can stop `vi` by sending an interrupt.

2. Displaying text and moving within the file

2.1 Arrow keys

The arrow keys on your keyboard move the cursor in `vi`. The `h`, `j`, `k`, and `l` keys also move the cursor.

`h` or ←	Move the cursor left a space. (The system *erase* key, usually DELETE, also works.)
`j` or ↓	Move the cursor down a line (in the same column).
`k` or ↑	Move the cursor up a line (in the same column).
`l` or →	Move the cursor right a space. (Space bar also works.)

You use these keys in command mode, and you can precede them with a number indicating how many spaces you want to move in the direction you want. For example, `5h` moves the cursor left `5` spaces.

2.2 Motion commands

Motion commands are either mnemonic, single-character commands or symbols that move the cursor in a file without affecting the file's contents in any way. With the exception of the `G` command, motion commands operate relative to the current cursor position. You can combine motion commands with a number (to indicate how many times the command executes) or with an operator, such as `d` for delete (to indicate how far the operation extends).

If preceded by a number ***n***, a motion command moves ***n*** motions (for example, ***n*** spaces or ***n*** lines) in that direction. The syntax is then

[*n*]*motion*

For example, to move the cursor three words forward, you would type `3w`.

Preceding these commands with an operator, such as `d` (delete) or `c` (change), indicates how far the operation of deleting or changing text extends. In this case, the syntax would be

[*n*][*opr*]*motion*

For example, to replace two words of text you would type `2cw`. See ''Combining Operators and Motions'' in this chapter for more details.

[*n*]`-`	Move the cursor to the beginning of the preceding line. Scroll if necessary.
[*n*]`+`	Move the cursor to the beginning of the next line. Scroll if necessary.
[*n*]`$`	Move the cursor to the end of the current line. Preceded by a number it means ''move to the end of the line *n* lines forward in the file.''
`^`	(Caret.) Move the cursor to the beginning of the first word on the line.
`0`	(Zero.) Move the cursor to the left margin of the current line.
[*n*]`|`	(Vertical bar.) Move the cursor to the beginning of the first column or to the column specified by *n*.
[*n*]`w`	Move the cursor to the beginning of the next word (or *n*th word).
[*n*]`W`	Move the cursor to the beginning of the next word (or *n*th word), ignoring punctuation.
[*n*]`b`	Move the cursor to the beginning of the preceding word (or *n*th word).
[*n*]`B`	Move the cursor to the beginning of the preceding word (or *n*th word), ignoring punctuation.

`[`*n*`]e`	Move the cursor to the end of the current word (or *n*th word).
`[`*n*`]E`	Move the cursor to the end of the current word (or *n*th word), ignoring punctuation.
`f`*x*	Move the cursor forward to the next instance of *x*, where *x* is a character.
`F`*x*	Move the cursor backward to the preceding instance of *x*, where *x* is a character.
`t`*x*	Move the cursor forward to one character position before the next instance of *x*, where *x* is a character.
`T`*x*	Move the cursor backward to one character position after the preceding instance of *x*, where *x* is a character.
`[`*n*`]G`	Move the cursor to the specified line number (Go to line number). `G` alone moves the cursor to the end of the file. `1G` moves to the beginning of the file.
`)`	Move the cursor to the beginning of the next sentence (defined as `.`, `!`, or `?` followed by two spaces or a newline character).
`(`	Move the cursor to the beginning of the current sentence.
`}`	Move the cursor to the beginning of the next paragraph. See "Moving by Text Block" for information on how a paragraph is defined.
`{`	Move the cursor backward to the beginning of a paragraph. See "Moving by Text Block" for information on how a paragraph is defined.
`]]`	(Right bracket, typed twice.) Move the cursor to the beginning of a new section. See "Moving by Text Block" for information on how a section is defined.
`[[`	(Left bracket, typed twice.) Move the cursor backward to the beginning of a section. See "Moving by Text Block" for information on how a

	section is defined.
`%`	Move the cursor to the matching parenthesis or brace. If you type `%` when the cursor is not on a parenthesis or brace, `vi` searches forward until it finds one on the current line and then jumps to the matching one.
[*n*]`H`	Move the cursor to the top-left position on the screen (Home) or the *n*th line from the top of the screen.
[*n*]`L`	Move the cursor to the bottom-left of the screen (Last) or the *n*th line from the bottom of the screen.
`M`	Move the cursor to the beginning of the middle line on the screen (Middle).
`` ` ` ``	(Back quote key typed twice.) Move the cursor back to where it was before the last absolute motion command. Absolute motion commands are those that move to a precise place (such as a line number or the word you searched for), not a place relative to the cursor position (such as CONTROL-d or `12j`).

Your file may have tab (CONTROL-i) characters in it. These characters are represented as several spaces expanding to a tab stop, where the default tab stop is eight spaces. When the cursor is at a tab, it sits on the last of the spaces representing that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may have nonprinting characters in it. These characters appear as a two-character code, with ^ as the first character; these two characters are treated as a single character.

2.3 Moving by text block

The parentheses, `(` and `)`, move the cursor to the beginning of the previous and next sentences, respectively. A sentence ends with a period, question mark, or exclamation mark, followed by either the end of the line or *two* spaces.

The braces, `{` and `}`, move the cursor over paragraphs. A paragraph begins after each empty line, at the `troff` request `.bp`, or at a paragraph macro. By default, this option uses `mm`'s paragraph macros (that is, the `.P` and `.LI` macros). You can change this default with the `set paragraphs` command; see Chapter 5, ''Using `ex`,'' in this manual. Each paragraph boundary is also a sentence boundary. You can precede the sentence and paragraph commands with a number to operate over groups of sentences and paragraphs.

`[[` and `]]` (left and right brackets, typed twice) move forward and backward over sections, respectively. Sections begin after each macro set in the `sections` option (normally `.H` and `.HU`) and each line with a form feed (CONTROL-l) in the first column. Section boundaries are always line and paragraph boundaries.

2.4 Go to a specific line

Typing `G` moves you to the end of your file. `vi` displays a tilde character (~) on each line past the last line of your text.

Type CONTROL-g to find out the current line number. This prints a message on the last line of your screen containing the name of the file you are editing, the current line number, the total number of lines in the buffer, and the percentage of the way through the buffer you are (in lines). You can also find your current line number by typing the colon character (`:`) to move to the `ex` command line and typing

```
.=
```

You can get back to the previous position in the file by typing a back quote twice (` `` `). Please refer to ''Line Selection'' in Chapter 5, ''Using `ex`.''

You can also move to a specific word or phrase in a file. To search for a particular string in your file, type the slash character (`/`) to move to the `ex` command line. You will see the slash character on the bottom line of your screen, waiting for you to specify the string you want to find. Type

/*string*

For more information about searching, please refer to ''Regular Expressions and Searching'' in Chapter 5, ''Using `ex`.''

2.5 Marking text

You can set up a special address with the `m`*x* command (where *x* is a letter between a and z) and then use this address anywhere you would use a `vi` address. After you have marked a line, you can refer to it by typing

`'`*x*

where *x* is the name you gave it.

2.6 Scrolling and paging

`vi` has several commands that scroll the text of a file up or down on your screen. You cannot combine these commands with other commands such as `d` (delete) or `c` (change). You can precede them with a number *n* indicating how many lines you want to move. The syntax for this is

[*n*]*scroll-command*

for example,

3CONTROL-b

You execute these commands by holding down the key labeled CONTROL while pressing the indicated letter.

[*n*]CONTROL-d	Move the cursor down half a screen (or by *n* lines).
[*n*]CONTROL-u	Move the cursor up half a screen (or by *n* lines).
[*n*]CONTROL-f	Move the cursor to the next screenful (or *n* screenfuls forward).
[*n*]CONTROL-b	Move the cursor to the previous screenful (or *n* screenfuls backward).
[*n*]CONTROL-e	Display another line at the bottom of the screen (or *n* lines).
[*n*]CONTROL-y	Display another line at the top of the screen (or *n* lines).
[*n*]CONTROL-p	Move the cursor to the previous line (or *n* lines backward, same column).
[*n*]<CR>	Move the cursor to the next line (or *n* lines forward, same column).
[*n*]`z`<CR>	Display the current full screen (or the full screen starting with *n*).

3. Inserting text

The following commands insert text. Note that you must press the ESCAPE key to terminate text insertions. Pressing ESCAPE returns you to command mode. You can always undo your last change by typing `u` in command mode.

Inserted text can contain newlines, but it does not have to.

`a`[*text*]ESCAPE	Insert *text* immediately after the cursor (append).
`A`[*text*]ESCAPE	Insert *text* at the end of the current line.
`i`[*text*]ESCAPE	Insert *text* immediately before the cursor (insert).
`I`[*text*]ESCAPE	Insert *text* at the beginning of the current line.
`o`[*text*]ESCAPE	Open a new line after the current line and insert *text* there (open).
`O`[*text*]ESCAPE	Open a new line before the current line and insert *text* there.

`i` places text to the left of the cursor, and `a` to the right; `I` inserts text at the beginning of the line, and `A` at the end of the line. Insert and append a few times to make sure you understand how this works. Make sure to press ESCAPE to terminate the text you insert.

You will often want to add new lines. Press `o`, which creates (opens) a new line after the line you are on. `O` creates a new line before the line you are on. After you create a new line, everything you type is inserted on the new line. Press ESCAPE to stop inserting.

To type in more than one line of text, press RETURN to end a line. This creates a new line and you continue typing. You can also set the `wrapmargin` option, which automatically moves your cursor to the following line once you have moved to a certain column. See Chapter 5, "Using `ex`," for more information.

3.1 Correcting text as you insert

The following characters correct text as you insert it (that is, while you're still in insert mode):

erase	The system erase character (often the ASCII backspace sequence CONTROL-h, DELETE, or #). Deletes the last input character.

kill	The system kill character (often CONTROL-u, CONTROL-x, or `@`). Deletes the current input line.
CONTROL-w	Delete the last word entered.

Your system kill character erases all the input on the current line. See *A/UX User Interface* to set your own kill and erase characters at login time.

Pressing CONTROL-h or your own erase character erases the last character you typed. Pressing CONTROL-w erases a whole word and leaves you after the preceding word.

4. Deleting text

The following commands delete text. You can precede most of these commands with a number indicating the extent of the command. For example,

```
3x
```

deletes three characters. Undo the last change by typing `u`, and repeat the last command by typing `.` (dot).

[*n*]`x`	Delete the character (or *n* characters), starting at the cursor.
[*n*]`X`	Delete the character (or *n* characters), backward from the character before the cursor.
`D`	Delete from the cursor to the end of the line.
[*n*]`d`*motion*	Delete one (or *n*) occurrences of the specified motion. You can use any of the true motion commands here. (See "Motion Commands" for more information.) For example, `3dw` deletes three words.
[*n*]`dd`	(`d` typed twice). Delete current line (or *n* lines including the current line).

You can transpose characters by `x`'ing the first character that is transposed and then typing `p`. For example, to correct the word `charcaters`, move the cursor to the transposed `c`, type `x` and then `p`. This deletes the `c` and then puts it in the proper place: `characters`.

You can also delete text by specifying the line numbers you want to delete; see "Deleting Text" in Chapter 5, "Using `ex`."

See "Recovering Lost Text" if you have deleted some text and want to get it back.

5. Changing text

The following commands replace text by simultaneously deleting the existing text and inserting new text. You can also precede these commands with a number, *n*, to indicate the extent of the command. For example, `4r` lets you replace four characters. Type `u` to undo these commands.

`r`*x*	Replace the character at the cursor with *x*. This is a one-character replacement. You don't need ESCAPE to terminate the command.
`R`[*text*]ESCAPE	Overwrite the characters on the screen with *text*. After you type `R`, whatever you type overwrites the existing text until you press ESCAPE.
[*n*]`s`[*text*]ESCAPE	Substitute character (or *n* characters) beginning at the cursor. `$` appears at the *n*th position in the text, so you know how much you are changing. Terminate with ESCAPE.
[*n*]`S`[*text*]ESCAPE	Substitute the entire current line (or *n* lines). `$` appears at end of the current line, or *n* lines are deleted before insertion begins. Terminate with ESCAPE.
[*n*]`c`*motion*[*text*]ESCAPE	Change *motion* to *text*, where *motion* is a motion command, for example, `w` for word(s), `}` for paragraph(s), `)` for sentence(s), and so on. You can also precede the commands with a number, *n*, to indicate the extent of the command. For example, `4cw`*text* lets you change four words and replace them with *text*. Terminate with ESCAPE.

[*n*]cc[*text*]ESCAPE	(c typed twice.) Change entire line (or *n* lines). Terminate with ESCAPE.
C[*text*]ESCAPE	Change from the cursor to the end of the line.

vi prints a message on the last line of the screen telling you how many lines you changed. vi also tells you when a change affects text you cannot see.

You can also change text by specifying line numbers on the ex command line; see "Changing Text" in Chapter 5, "Using ex."

5.1 Combining operators and motions

You make larger changes by combining operators (d for delete, c for change, s for substitute, and so on) with the motion commands introduced earlier: w for word(s),) for sentence(s), } for paragraph(s), /*pattern* for context search(es), and so on. The syntax for the general case is

operator motion-command

Move to the beginning of a word and type dw to delete a word. Now try db; this deletes the word *to the left* of the cursor. The command

```
d}
```

deletes the text from your current cursor position to the next paragraph delimiter—a blank line or an nroff/troff command for list or paragraph.

The command

```
d)
```

deletes the rest of the current sentence. Similarly, d(deletes the line to the left of the cursor. d(deletes the preceding sentence if you are at the beginning of the current sentence, or the current sentence up to the cursor position if you are not at the beginning of the sentence.

You can also use these operators with the / (or ?) search command to change similar phrases in a document. For more information, please refer to "Regular Expressions and Searching."

Another useful operator is `c` (for change). `cw` changes a single word to the text you insert. Press ESCAPE to terminate. Move to the beginning of a word and type

`cw`*new-word*

(followed by ESCAPE). Notice that the end of the text to change was marked with the dollar sign character (`$`).

`f` and `t` are useful with operators like `c` (change) and `d` (delete) to change a section of text that is not recognized as a delimited word.

5.2 Undoing the last command

`u`	Undo the last command, including a preceding undo command.
`U`	Undo changes to the current line.

If you make an incorrect change (whether large or small), use the `u` (undo) command to undo it. Notice that `u` also undoes the previous `u`.

The undo command reverses only a single change. After you make several changes to a line, you may decide that you would rather have the original line back. `U` restores the current line to the way it was before you started changing it, but only if you have not left the line before pressing `U`.

If you have made several changes deleting text, you can use `u` to undo only your last change. You can still recover deleted text, however, even if it is too late to use the `u` command. See ''Recovering Lost Text'' later in this chapter.

5.3 Repeating the last command

`.`	Repeat the last command that changed the buffer.
`n`	Repeat the last / or `?` search command (next).
`N`	Repeat the last / or `?` search command in the opposite direction.
[*n*]`;`	Repeat the last `f`, `F`, `t`, or `T` command (once or *n* times).

[*n*]`,`	Repeat the last `f`, `F`, `t`, or `T` command in the opposite direction (once or *n* times).
`&`	Repeat the last single substitution.

For more information, please refer to "Changing Text" in Chapter 5, "Using `ex`."

5.4 Storing text in named buffers

The editor has a set of buffers named `a` through `z`. If you precede any delete or replacement command with

`"`*a*

(where the double quote character indicates "buffer name" and *a* is any single lowercase character), that named buffer will contain the text deleted by the command. For example,

```
"a3dd
```

deletes three lines, starting at the current line, and puts them in register `a`.

Move the cursor to where you want the lines and type

```
"ap
```

or

```
"aP
```

that is, "put contents of register `a`," to put them back.

6. Copying and moving text

The following commands "yank" text (duplicate it in a buffer) and "put" it at another location in the text. In the following, *buf-spec* is the `"`*a* buffer notation.

[*n*][*buf-spec*]`y` *motion*	Yank the specified object (word, paragraph, and so on) or *n* objects into a buffer.
[*n*][*buf-spec*]`yy`	Yank the current line (or *n* lines) into a buffer.
[*n*][*buf-spec*]`Y`	Equivalent to `yy`.
[*buf-spec*]`p`	Put the contents of the buffer in the text after the cursor. Lines you yank are placed on new lines

following the current line. Other objects, such as words or paragraphs, are inserted immediately following the cursor.

[*buf-spec*]`P`	Put the contents of the buffer in the text before the cursor. Lines you yank are placed on new lines preceding the current line. Other objects, such as words or paragraphs, are inserted immediately preceding the cursor.

`vi` has a single unnamed buffer where it saves the last text you deleted or changed, and a set of named buffers (`a` through `z`) where you can save copies of text and move text around in your file and between files. For more information on these named buffers, see "Storing Text in Named Buffers."

`y` yanks a copy of the specified object into the unnamed buffer. For example, `y3w` puts three words in the buffer.

You can then put the text 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 is part of a line or partially spans more than one line, the text is put back after the cursor (or before it, if you use `P`). If the yanked text is whole lines, they are put back as whole lines, without changing the current line. This acts much like an `o` or `O` command.

Try `YP`. This makes a copy of the current line and places it before the current line. `Y` is a convenient abbreviation for `yy`. `Yp` copies the current line and places it after the current line. You can give `Y` a count of lines to yank and duplicate several lines. Try typing `3YP`.

You can also copy and move text by specifying line numbers on the `ex` command line; see "Copying and Moving Text" in Chapter 5, "Using `ex`."

6.1 Recovering lost text

In addition to the named buffers (`a-z`) and the unnamed buffer (the "undo" buffer), there are nine numbered buffers where the editor places each piece of text you delete (or yank).

The most recent deletion (or yank) is in the undo buffer and also in buffer 1. The next most recent deletion or yank is in buffer 2, and so on. Each new deletion pushes down all older deletions; those older than 9 disappear. If you delete lines and then regret it, you can get the *n*th previous deleted text back in your file using the command

```
"np
```

where *n* is register 1 through 9. The double quote character (") here means "buffer number," *n* is the number of the buffer (use the number 1 for now), and `p` is the put command, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, type `u` to undo this, and then a period (`.`) to repeat the put command. In general, the period (`.`) repeats the last change you made. When the last command refers to a numbered text buffer, `.` increments the number of the buffer before repeating the command. For example, typing

```
"1pu.u.u.u.u.u.u.u.
```

shows you all the deleted text that has been saved (nine deletions). You can omit the `u` commands here to gather all this text in the buffer, or you can stop after any `.` command to keep only the text recovered so far. You can use `P` instead of `p` to put the recovered text before rather than after the cursor.

You can use the `ex` commands `co` and `m` to copy and move text, respectively. For more information, please consult Chapter 5, "Using `ex`."

7. Regular expressions and searching

You can search for words or phrases in files by typing a slash (/), followed by the phrase you want to find, followed by RETURN. For example, typing

```
/grump<CR>
```

takes you to the next instance of `grump` in the file. This is an `ex` command. When you type the slash, the cursor moves to the `ex` command line. You can also combine searching with commands in a file. For more information about this and other aspects of searching, please refer to "Regular Expressions and Searching" in Chapter 5.

"Using `ex`," in this manual.

Regular expressions use special characters and notation to specify a set of character strings. For example, the regular expression `.` (dot) matches *any* single character. Therefore, the search command

```
/c.t
```

takes you to the next occurrence of all words containing a `c` followed by any character followed by a `t` (`cat`, `cbt`, `cct`, and so on).

Regular expressions can be extremely useful when you're editing a file. For more information about them, please refer to "Regular Expressions and Searching" in Chapter 5, "Using `ex`," in this manual.

8. Working with multiple files

You can edit more than one file at the same time. To open more than one file at once, enter `vi` with the command

`vi` *file1 file2*

file1 will be opened first. When you write the file, you can move to the second file by typing the colon character (`:`) to move to the `ex` command line and typing

```
n
```

You must write the first file before trying to get into the second file or `vi` won't let you move.

To return to the first file, type

CONTROL-^

(to get this sequence on many terminals you must type CONTROL-SHIFT-6).

You can also read the contents of another file into the buffer by typing the colon character (`:`) to move to the `ex` command line and typing

`r` *filename*

A copy of *filename* will be inserted after the current line. See "Copying Another File to the Current Buffer" in Chapter 5, "Using `ex`," for more information.

For more information about editing several files at the same time, please refer to "Working With Multiple Files" in Chapter 5, "Using ex."

9. Using shell commands in vi

There are several ways of interacting with the shell from within vi. All of these require that you type the colon character (:) to move to the ex command line and then type the appropriate ex command. Please refer to "Using Shell Commands in ex" in Chapter 5, "Using ex."

10. Setting options

You can customize your vi environment by setting various ex options. To set an option in vi type the colon character (:) to move to the ex command line and then type the appropriate set command. The ex options that are especially useful in vi are described briefly below.

set wi[ndow]=*n*	Changes the number of lines in your editing window.
set scr[oll]=*n*	Changes the number of lines you scroll through with the CONTROL-d command.
set para[graphs]=*macro-strings*	Changes the strings vi searches for when you press { or }. Valid strings are the paragraph and list macros from the mm and ms macro packages.
set sect[ions]=*macro-strings*	Changes the strings vi searches for when you press [[or]]. Valid strings are the section macros from the mm and ms macro packages.
set redraw	Forces a dumb terminal to redraw the characters to the right of the cursor as you enter text in vi.
set w[rap]m[argin]=*n*	Sets the column where the cursor automatically returns to the left margin.

`set nomesg`	Prevents messages from other people while in `vi`.

For more information about setting options and a list of all the available options, please refer to "Setting Options" in Chapter 5, "Using `ex`."

11. Mapping and abbreviations

Mapping and abbreviation are available as a joint facility of `ex` and `vi`. The mapping or abbreviation must be defined on the `ex` command line, ***but is useful only in visual mode.*** Since `vi` is required for these to work, the examples are structured to work within `vi`. These commands require that you type the colon character (`:`) to move to the `ex` command line before typing the appropriate command.

11.1 Mapping: **map**

In `vi` you can use the `ex` command `map` to create a macro. A macro sets a string (usually a single key) equal to a command or sequence of commands.

The general format of the `map` command in `vi` is

`:map` *string definition*

(followed by RETURN), where *string* is usually a single keystroke or function key, and *definition* is the map definition representing a command or sequence of commands.

After you enter a `map` command, typing *string* performs the specified command. *string* can't be more than 10 characters and *definition* can't be more than 100 characters. If it takes longer than a second to type *string*, however, `vi` times you out before recognizing the string (you can prevent this by setting the option `notimeout`; see "Setting Options" in Chapter 5, "Using `ex`").

When you set up a `map` from within `vi`, it lasts only as long as your current editing session. If you're editing several files in one session (by entering several names on the command line) and you use the `:n` command to edit them, however, the `map` definitions hold for all files until you quit the editor. To make a more permanent definition, insert it in the `EXINIT` variable in your `.profile` (or `.login`) file or in `$HOME/.exrc` (see "Setting Options").

CONTROL-v allows you to insert nonprinting characters (<CR>, blanks, tabs, control sequences) in your `map` definition. Pressing RETURN ends the `map` command, so to include a <CR>, ESCAPE, or any other nonprinting character in *definition*, you must escape it with CONTROL-v.

For example, to make the character `q` write and exit the editor, type the command

```
:map q :wqCONTROL-v<CR><CR>
```

This maps the sequence

```
:wq<CR>
```

into the character `q`, so that when you type `q` in `vi` command mode, the command `:wq`<CR> executes. Without CONTROL-v, the first RETURN would have ended the `map` command, rather than becoming part of the `map` definition. With CONTROL-v, however, the first RETURN is part of the *definition*, and the second ends the `map` command.

You can use # in the `map` command to represent function keys. Some terminals don't have function keys, but most terminals do. If the *string* is `#0` through `#9`, it maps to the corresponding function key, *not* the two-character sequence #*n*. (Note that on the Apple keyboard for the Macintosh II, #*n* refers to the corresponding key on the 10-key pad.)

You can also use the `map` command so that one key calls a second key, which calls the first one again. This is useful to repeat an editing action throughout an entire file. When you're writing these double maps that call each other, however, *make sure the pattern you are searching for changes as each command executes.*

When you `map` a key to the function, try to choose a key that does not already have a function you need to use. For example, if you have a file containing names and telephone numbers in one format, and you want to change it globally as follows:

Initial Format	New Format
`...`	`...`
`Kent, Clark`	`Kent, Clark 123-4567`
`123-4567`	`Wayne, Bruce 567-8910`
`Wayne, Bruce`	
`567-8910`	
`...`	`...`

map the keys `g` and `h` as follows:

```
:map g /^[0-9]/CONTROL-v <CR>kJh
:map h /^[0-9]/CONTROL-v <CR>kJg
```

Then, in `vi`'s command mode, typing `g` or `h` executes the command

```
/^[0-9]/CONTROL-v <CR>
```

That is, the search command looks for a line beginning with a number 0 through 9. You must follow this command by pressing RETURN to execute it, and you must escape the RETURN in this definition with CONTROL-v. When the cursor is positioned on a line beginning with a number 0 through 9, the command

```
kJh
```

indicates "go up one line" (`k`), "join this line and the next line" (`J`), and "call `h`," which repeats this sequence of commands and calls `g`.

These commands finish when there are no lines left in the file that begin with a number 0 through 9. The number of lines *beginning* with a number is reduced by one each time the `J` command executes. Otherwise, these commands can go into an infinite loop, calling each other indefinitely. If this happens, send an *interrupt*.

You can delete macros with

`:unmap` *string*

The `undo` command reverses the effect of the entire macro as a unit.

11.2 Abbreviations

You can define a short string that expands to a longer string in the text. The commands to perform this are `abbreviate` and `unabbreviate` (`ab` and `una`) and have the syntax

:*abbreviation-command wd* [*word*]

(followed by RETURN), where *abbreviation-command* is `ab[breviate]` or `una[bbreviate]`, *wd* is the abbreviation you're defining, and *word* (only applicable for `abbreviate`) is the string it represents. Note that the order of the arguments is the reverse of what you might expect.

wd can't be more than 10 characters and *word* can't be more than 100 characters. If it takes longer than a second to type *wd*, however, `vi` times you out before recognizing the string (you can prevent this by setting the `ex` option `notimeout`; see ''Setting Options'' in Chapter 5, ''Using `ex`'').

After you enter an abbreviate command, typing *wd* translates it into *word* immediately. For example, after you type the command

```
:ab cs Department of Computer Sciences
```

enter (in insert mode) the string `cs` (followed by spaces, a newline character, or punctuation marks). This expands immediately to the phrase ''Department of Computer Sciences,'' which is entered as part of your `vi` buffer. If you type `cs` as part of a larger word, however, it is left alone. The abbreviation echoes as you type it, and when it reaches a delimiter that sets it apart as a single word, it immediately expands into the longer string.

12. Other features

12.1 Escaping nonprinting characters in `vi`

If you need to enter a control character such as the system *erase* or *kill* character on your screen when you are inserting text, you need to ''escape'' it so `vi` will not interpret its meaning. You can do this as follows:

\	If the character or sequence prints on your screen but has a meaning to the editor that you want to prevent (for example, `$`), precede the character with a backslash.

CONTROL-v	If the character or sequence is invisible on your screen (for example, CONTROL-l), you need to escape its meaning to the editor by preceding it with CONTROL-v.

For example, to type an ESCAPE character into your file type CONTROL-v before you press the ESCAPE key. The CONTROL-v prints a caret (`^`) at the current cursor position, indicating that the editor expects you to type a control character. When you press the ESCAPE key, you will see `^[` in your file. After CONTROL-v, you can insert any nonprinting character except the null (`@`) character or the line-feed (CONTROL-j) character into the file. Using CONTROL-v is the only way to insert CONTROL-s or CONTROL-q.

12.2 Saving text and exiting

It is recommended that you periodically save the text you have entered in the buffer using the line-mode command

```
    :w
```

This command writes the text into a permanent file, overwriting any previous version of the file. See ''Saving Text and Exiting'' in Chapter 5, "Using `ex`."

You can exit `vi` by making sure you are in command mode (press the ESCAPE key) and giving the command

```
    ZZ
```

This command writes the changes you made to the buffer back into the file you were editing and quits `vi`. However, if you did not change to the buffer, this command does not force a write operation.

If you want to force a write operation (this is especially recommended when recovering a file using `vi -r`), use the `ex` comand

```
    :wq
```

There are several `ex` commands for exiting `vi` in different ways. To use them type the colon character (`:`) to move to the `ex` command line and type the appropriate `ex` command. For more information see ''Saving Text and Exiting'' in Chapter 5, ''Using `ex`.''

13. Error conditions

13.1 Redrawing the screen

Occasionally the screen will need to be refreshed (for example, a program can write to the screen, or line noise can jumble up the screen).

Press CONTROL-l to redraw the screen. On certain terminals, CONTROL-r works instead.

You can redraw the screen so that a specified line shows at the top, middle, or bottom. To do this, move the cursor to that line and type

```
    z
```

Pressing RETURN after the `z` command redraws the screen with the line at the top; a period (`.`) after the `z` places the line at the center; a minus sign (–) after the `z` places it at the bottom.

13.2 When your system is slow

You can reduce the overhead of refreshing the screen for each change, scroll, and so on, by limiting the window size. You can invoke `vi` with a specific window size with the command

`vi -w`*n* *filename*

where *n* is a number less than 23. For example,

`vi -w3` *filename*

This sets the initial size of the window to three lines, and allows the window to expand as you add lines.

You can control the size of the window that `vi` redraws when the screen clears by specifying a window size as an argument to any of the following commands:

```
    :   /   ?   [[   ]]   `   '
```

If you search for a string in a file, preceding the first search command with a small number (for instance, 3) draws three-line windows around each instance of the string.

You can easily expand or contract the window, placing the current line where you want, by giving a number after the `z` command and before the following RETURN, period (`.`), or minus (–). For example, the command

```
z5.
```

redraws the screen with the current line in the center of a five-line window. The command `5z.` has an entirely different effect, placing line 5 in the center of a new window.

If `vi` is updating large portions of the display, you can interrupt it by sending an interrupt signal (usually DELETE or CONTROL-c). This may partially confuse `vi` about what is displayed on the screen. You can clear up the confusion by typing CONTROL-l or by moving or searching again, ignoring the current state of the display.

See ''Terminal Characteristics Unknown: Open Mode'' for another way to use the `vi` command set on slow terminals.

13.3 Large files: out of temp filespace

`vi` prints the message

```
Out of temp filespace
```

when it doesn't have enough buffer space (in `/tmp`) to hold the file. It might refuse to open the file when it prints this, or it might open it and load only part of the file into the buffer. The latter is dangerous because if you write the file when only half of it is in the buffer, you can lose the other half.

The best thing to do when that message prints is to get out of `vi` by typing the colon character (`:`) to move to the `ex` command line and typing

```
q!
```

Then go up to `/tmp` and delete any files that aren't necessary. After you've deleted some files, go back and try opening your document again.

If your file is very large, you may have to use the A/UX `split` command to break it into smaller text files before you can use `vi` to edit it. See `split(1)` in the *A/UX Command Reference* for more information.

13.4 Terminal characteristics unknown: open mode

`vi` uses the value of the TERM environment variable and the terminal description file `terminfo` to control the screen. If it does not find a description of your terminal in `terminfo`, it displays the message

```
I don't know what kind of terminal you are on
 —all I have is 'unknown'

[Using open mode]
```

Sometimes this message indicates an incorrect value in the TERM variable. Type

```
   echo $TERM
```

to find out the current value of TERM, and see *A/UX User Interface* to reset this variable.

In open mode, the editor uses a single-line window into the file and displays a new line that is always below the current line. Two `vi` commands, `z` and CONTROL-r, work differently in open mode. 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 hard-copy terminal, CONTROL-r prints the current line. On these 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 the second is the copy you are working on. When you delete characters, `vi` prints backslashes through the deleted characters. The editor also reprints the current line soon after these changes so that you can see what the line looks like again.

Open mode can be useful on very slow terminals.

13.5 Recovering lost files

If the system crashes or you receive a hangup signal on a dial-in line, you can recover your work even if you did not write the changes to the file. Move to the directory you were in when the system crashed, and give the command

`vi -r`*filename*

(the `-r` flag option for recover) where *filename* is the file you were editing when the system went down. `vi -r` will tell you if there are any files to be recovered.

In some cases, a few lines of the file may be lost. These lines will almost always be the last few you changed.

> *Note:* It is not advisable to exit a file that you have just recovered using the `ZZ` command, since `ZZ` does not guarantee that the file will be written. After you have checked the recovered file, you should force a write using `:w`.

14. Command summary

14.1 Editing commands

The following commands are used in `vi`'s command mode.

Command	Description
[*n*]CONTROL-b	Move the cursor to the previous screen (or *n* screens backward).
[*n*]CONTROL-d	Move the cursor down half a screen (or by *n* lines).
[*n*]CONTROL-e	Display another line at the bottom of the screen (or *n* lines).
[*n*]CONTROL-f	Move the cursor to the next screen (or *n* screens forward).
CONTROL-g	Give current line number and filename and the percentage along in the file.
CONTROL-h	Move the cursor back one space.
[*n*]CONTROL-j	Move down one line in the same column (or *n* lines).

Command	Description
CONTROL-l	Redraw screen (certain terminals only – others use CONTROL-r).
[*n*]CONTROL-m	Same as pressing the RETURN key.
[*n*]<CR>	Move the cursor to the next line (or *n* lines forward, first column).
[*n*]CONTROL-p	Move the cursor to the previous line (or *n* lines backward, same column).
CONTROL-r	Redraw screen (most terminals – others use CONTROL-l).
[*n*]CONTROL-u	Move the cursor up half a screen (or by *n* lines).
[*n*]CONTROL-y	Display another line at the top of the screen (or *n* lines).
CONTROL-^	Edit the alternate file. See ''Working With Multiple Files'' for more information
`a`[*text*]ESCAPE	Insert *text* immediately after the cursor (append).
`A`[*text*]ESCAPE	Insert *text* at the end of the current line.
[*n*]`b`	Move the cursor to the beginning of the preceding word (or *n*th word).
[*n*]`B`	Move the cursor to the beginning of the preceding word (or *n*th word), ignoring punctuation.

Command	Description
[*n*]c*motion*[*text*]ESCAPE	Change *motion* to *text*, where *motion* is a motion command.
cc[*text*]ESCAPE	(c typed twice.) Change entire line (or *n* lines).
C[*text*]ESCAPE	Change from the cursor to the end of the line.
[*n*]d*motion*	Delete one (or *n*) occurrences of the specified motion. You can use any of the true motion commands here. (See ''Motion Commands'' for more information.) For example, 3dw deletes three words.
[*n*]dd	(d typed twice.) Delete current line (or *n* lines including current line).
D	Delete from the cursor to the end of the line.
[*n*]e	Move the cursor to the end of the current word (or *n*th word).
[*n*]E	Move the cursor to the end of the current word (or *n*th word), ignoring punctuation.
[*n*]f*x*	Move the cursor forward to the first instance of *x* (*n* specifies the *n*th instance).
[*n*]F*x*	Move the cursor backward to the first instance found of *x* (*n* specifies the *n*th instance).

Command	Description
[*n*]G	Move the cursor to the specified line number (Go to line number). G alone moves the cursor to the end of the file. 1G moves to the beginning of the file.
h or ←	Move the cursor left a space. (BACKSPACE also works.)
[*n*]H	Move the cursor to the top-left position on the screen (Home). or the *n*th line from the top of the screen.
i[*text*]ESCAPE	Insert *text* immediately before the cursor (insert).
I[*text*]ESCAPE	Insert *text* at the beginning of the current line.
[*n*]j or [*n*]↓	Move the cursor down a line (in the same column) or down *n* lines.
[*n*]J	Join current line with next line or the next *n* lines.
[*n*]k or [*n*]↑	Move the cursor up a line (in the same column) or up *n* lines.
[*n*]l or [*n*]→	Move the cursor right a space or *n* spaces (Space bar also works).
[*n*]L	Move the cursor to the bottom-left of the screen (Last) or the *n*th line from bottom of the screen.

Command	Description
m	Mark the current position of the cursor in the register specified by the following letter (a through z). Return to this position with ` and the register letter.
M	Move the cursor to the beginning of the middle line on the screen (Middle).
n	Repeat the last / or ? search command (next).
N	Repeat the last / or ? search command in the opposite direction.
o[*text*]ESCAPE	Open a new line after the current line and insert *text* there (open).
O[*text*]ESCAPE	Open a new line before the current line and insert *text* there.
[*buf-spec*]p	Put the contents of the buffer in the text after the cursor. Lines you yank are placed on new lines following the current line. Other objects, such as words or paragraphs, are inserted immediately following the cursor. *buf-spec* specifies a buffer "a through "z.
[*buf-spec*]P	Put the contents of the buffer in the text before the cursor. *buf-spec* specifies a buffer "a through "z.
Q	Quit vi and go to ex command line. (See :vi for returning to vi.)

Command	Description
[*n*]`r`*x*	Replace the character (or *n* characters) at the cursor with *x*. This is a one-character replacement. You don't need ESCAPE to terminate the command.
`R`[*text*]ESCAPE	Overwrite the characters on the screen with *text*. After you type `R`, whatever you type overwrites the existing text until you press ESCAPE.
[*n*]`s`[*text*]ESCAPE	Substitute character (or *n* characters) beginning at the cursor. `$` appears at the *n*th position in the text, so you know how much you are changing.
[*n*]`S`[*text*]ESCAPE	Substitute the entire current line (or *n* lines). `$` appears at the end of the current line, or *n* lines are deleted before insertion begins.
[*n*]`t`*x*	Move the cursor forward to just before the first instance of *x* (or the *n*th instance).
[*n*]`T`*x*	Move the cursor backward to just before the first instance of *x* (or the *n*th instance).
`u`	Undo the last command, including a preceding undo command.
`U`	Undo changes to the current line.
[*n*]`w`	Move the cursor to the beginning of the next word (or *n*th word).
[*n*]`W`	Move the cursor to the beginning of the next word (or *n*th word), ignoring punctuation.

Command	Description
[*n*]x	Delete the character (or *n* characters), starting at the cursor.
[*n*]X	Delete the character (or *n* characters), backward from the character before the cursor.
[*n*][*buf-spec*]y *motion*	Yank the specified object (word, paragraph, and so on) or *n* objects into a buffer. *buf-spec* specifies a buffer "a through "z.
[*n*][*buf-spec*]yy	Yank the current line (or *n* lines) into a buffer. *buf-spec* specifies a buffer "a through "z.
[*n*][*buf-spec*]Y	Equivalent to yy. *buf-spec* specifies a buffer "a through "z.
[*n*]z	Display the current full screen (or the full screen starting with *n*). Change the placement of the current line by following z with one of these characters: <CR> place current line at the top of the screen. . place current line at the center of the screen. – place current line at the bottom of the screen.
ZZ	Quit vi, performing a write operation first if there were changes made to the file.
–	Move the cursor to the beginning of the preceding line. Scroll if necessary.

Command	Description
+	Move the cursor to the beginning of the next line. Scroll if necessary.
[*n*]$	Move the cursor to the end of the current line. Preceded by *n* it means "move to the end of the line *n* lines forward in the file."
^	Move the cursor to the beginning of the first word on the line.
0	(Zero.) Move the cursor to the left margin of the current line.
[*n*] |	(Pipe or vertical bar.) Move the cursor to the beginning of the first column or to the column specified by *n*.
)	Move the cursor to the beginning of the next sentence (defined as `.`, `!`, or `?` followed by two spaces or a newline character).
(	Move the cursor to the beginning of the current sentence.
}	Move the cursor to the beginning of the next paragraph (the default definition of a new paragraph is `.P`, `.LI`, or `.bp`) or to the next blank line.
{	Move the cursor backward to the beginning of a paragraph (the default definition of a new paragraph is `.P`, `.LI`, or `.bp`) or to the last blank line.

Command	Description
`]]`	(Right bracket, typed twice.) Move the cursor to the beginning of a new section (default definition is by `.H` or `.HU`).
`[[`	(Left bracket, typed twice.) Move the cursor backward to the beginning of a section (default definition is by `.H` or `.HU`).
`%`	Move the cursor to the matching parenthesis or brace. If you type `%` when the cursor is not on a parenthesis or brace, `vi` searches forward until it finds one on the current line and then jumps to the matching one.
` `` `	(Back quote key typed twice.) Move the cursor back to where it was before you used the last absolute motion command. Absolute motion commands are those that move to a precise place (such as a line number, or the word you searched for), not a place relative to the cursor position (such as CONTROL-d or `12j`).
`.`	Repeat the last command that changed the buffer.
[*n*]`;`	Repeat the last `f`, `F`, `t`, or `T` command (once or *n* times).
[*n*]`,`	Repeat the last `f`, `F`, `t`, or `T` command in the opposite direction (once or *n* times).
`&`	Repeat the last single substitution.

14.2 Insert mode commands

The following commands are used in `vi`'s insert mode.

Command	Description
CONTROL-d	During an insert, backtabs over `autoindent` white space at the beginning of a line.
CONTROL-i	Input tab.
CONTROL-q	Escape a single character or control sequence by preceding it with CONTROL-q.
CONTROL-t	Inserts a `shiftwidth` wide white space if pressed at the beginning of a line with `autoindent` set.
CONTROL-v	Escape a single character or control sequence by preceding it with CONTROL-v.
CONTROL-w	Delete the last word entered.
erase	The system erase character (often DELETE, CONTROL-h, or `#`). Deletes the last input character.
kill	The system kill character (often CONTROL-u, CONTROL-x, or `@`). Deletes the current input line.

Chapter 5
Using **ex**

Contents

Chapter 5

Using ex

1. ex basics

Once you have invoked `ex`, you enter commands at `ex`'s prompt, the colon (`:`). `ex` commands are words (such as `write` or `edit`), which you can abbreviate. A complete list of commands and their abbreviations appears at the end of this chapter.

1.1 Starting ex

Usually, you start `ex` with the following command:

`ex` *filename* ...

where *filename* is the name of the file to edit.

1.1.1 ex command syntax

The command to invoke `ex` is

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

The options are as follows:

-	(Minus sign.) Suppress interactive feedback. This is useful when you write shell scripts that use the `ex` editor.
`-v`	Equivalent to using `vi`.
`-t` [*tag*]	Start editing the file at *tag* (usually a spot marked with the `ctags` program). Equivalent to an initial `tag` command. This is described under ''Editing Programs.''
`-r` [*filename*]	Recover a file after an editor or system crash; if you don't specify *file*, it lists the saved files.
`-l`	Set the `showmatch` and `lisp` options for editing LISP programs. These are described under ''Setting Options.''

`-w`*n*	Set the window size to *n* lines.
`-x`	Prompt for a *key* to encrypt and decrypt the file (see `crypt(1)` in *A/UX Command Reference*). The file should already be encrypted using the same key.
`-R`	Set the `readonly` option, making it impossible to write the file with the `write` command.
`+[`*command*`]`	Move to the line specified by *command* where *command* is either a regular expression (see "Regular Expressions and Searching") or a line number (for example, `+100` starts editing at line 100). If you omit *command,* `ex` moves the cursor to the last line of the first file.
filename	The file(s) to edit.

1.1.2 ex initialization

When you start `ex`, it sets up your editing environment with the following steps:

- reads the `TERM` variable to find out what terminal you're using
- sets any options you've specified in the `.exrc` file in the current directory or your home directory
- sets any options you've specified in the `EXINIT` variable (usually set in the `.profile` (or `.login`) file in your home directory)

You can set the same options with either the `.profile`, `.login`, or `.exrc` files. The options are described in "Setting Options" later in this chapter.

1.2 Opening a file

To create and "open" a new file (or open an existing file) in `ex`, type

`ex` *filename*

where *filename* is the name of the file you're creating (or opening). For example, to open the file `sheep`, type

```
ex sheep
```

When you use ex to create a new file, ex opens some temporary storage space that is referred to as the **buffer.**

When you edit an existing file, ex places a copy of that file in the buffer. Changes you make to the text in the buffer (for example, to a copy of the sheep file mentioned above) are made only to this temporary copy. ex does not change the contents of the file until you write the file (see ''Saving Text and Exiting'').

Note: You should periodically write your changes to the file to prevent losing material if the system crashes or is interrupted.

1.3 ex modes

ex has a number of modes:

- **Command mode** is ex's primary mode; that is, when you invoke ex you are initially in command mode and everything you type is interpreted as a command. In command mode, you enter commands at the colon (:) prompt and terminate them with a RETURN character.
- In **input mode,** ex assumes that what you type is text (rather than commands), and it doesn't display a prompt. Invoking the append, insert, or change commands in command mode places you in input mode. Resume command mode by typing a period (.) at the beginning of a line and pressing RETURN.
- You can also enter open mode from ex using the o command. **open mode** allows you to use vi commands, but limits your movement to within one line at a time. (It is like visual mode with a screen one line long.) This is convenient if you want to use vi commands on a dumb terminal. (A smart, or addressable-cursor, terminal is required for you to use vi itself.) Type Q to return to ex.
- When you invoke ex with the command ex -v you enter the editor in its visual mode. **visual mode** allows movement and editing throughout the displayed screen of text. See ''Switching to Visual Mode.''

1.4 Switching to visual mode

When you invoke `ex` you can switch to `vi` by giving the command

```
vi
```

at the `ex` colon prompt on the bottom line of your screen. This invokes visual mode and places the current line as the first line on the screen. To return to `ex` from visual mode, type

```
Q
```

1.5 Special keys

RETURN

The RETURN (carriage return) key terminates all `ex` commands.

The examples in this chapter assume that you press RETURN after all commands unless shown otherwise.

The *interrupt* key

The *interrupt* key (set to CONTROL-c in the A/UX standard distribution) sends an interrupt signal to the editor. It is a forceful way of stopping `ex` from executing a command after you have pressed RETURN.

2. Displaying text and selecting lines

In `ex` you can display text on the screen by specifying a line number or range of line numbers followed by the `print` command. The next section describes how to specify which lines you want to select.

2.1 Line selection

You can prefix most commands with addresses. These addresses tell `ex` which lines to perform the command on. For example, `10print` prints the tenth line in the buffer.

Here are a few basic rules to follow when using line addresses in `ex`.

- Commands that don't require an address (such as the `quit` command to leave `ex`) regard an address as an error. The command summary at the end of this chapter specifies which commands need addresses and which don't.

- For `ex` commands that require an address, `ex` assumes a default address if you don't supply one. If you give more addresses than a command requires, ex uses the last one or two, depending on the command being attempted.

- For ex commands requiring two addresses, the second address must follow the first address. If you use two addresses, you can separate them with a comma (,) or a semicolon (;). Using , calculates both addresses relative to the current line. For example, if you are on line 1,

  ```
  +2,+4print
  ```

  prints lines 3 through 5. Using ; calculates the second address using the first address as the current line. For example, if you are on line 1,

  ```
  +2;+4print
  ```

  prints lines 3 through 7.

Throughout the rest of this chapter, the term *lineno* (line number) denotes an expression that identifies a single line in a file, numbered with *lineno*. This includes a search for a pattern (see /*pattern*[/], below).

Throughout the rest of this chapter, the term *line-spec* (line specifier) denotes an expression that identifies zero or more lines in a file. A valid *line-spec* can be a single line number *lineno;* a range of line numbers *line1*, *line2*; a context search resulting in zero or more lines (see "Regular Expressions and Searching"); or a regular expression resulting in zero or more lines (see "Regular Expressions and Searching"). Commonly used abbreviations include $ (the last line of a file) and . (the current line). In the following commands, *line-spec* defaults to the current line unless stated otherwise.

A *line-spec* may consist of any one of the following expressions:

.	Dot (.) indicates the current line. This is the default address for most commands. Most commands leave the current line as the last line they affect.
line1, *line2*	A range of line numbers beginning with *line1* and ending with *line2*.

lineno	Move to *lineno*. You can find out the current line number by typing ``.=''.
`$`	The last line in the buffer.
`%`	An abbreviation for `1,$`; the entire buffer.
+[*n*]	Forward *n* lines from the current line. `+3` and +++ are equivalent; if the current line is line 100, they address line 103.
-[*n*]	Backward *n* lines from the current line. `-3` and --- are equivalent; if the current line is line 103, they address line 100.
/*pattern*[/]	Forward to a line containing the regular expression *pattern* (see ``Regular Expressions and Searching'' later in this chapter). `ex` searches forward until it reaches the end of the file, then it searches from the first line of the file to where you began your search. If you want to print the next line containing *pattern*, you don't have to include the trailing `/`. `/` is shorthand for ``search forward for the last pattern you scanned for.''
?*pattern*[?]	Backward to a line containing the regular expression *pattern* (described under ``Regular Expressions and Searching''). `ex` searches backward until it reaches the beginning of the file, then it searches from the last line of the file to where you began your search. If you want to print the next line containing *pattern*, you don't have to include the trailing `?`. `?` is shorthand for ``search backward for the last regular expression you scanned for.''
`` ` ` ``	(Back quote typed twice.) Refers to your position before the last absolute motion (an **absolute motion** specifies the line to move to, while a **relative motion** specifies the distance to move from the current line.)

`*x*	(Where *x* is a letter from *a* to *z*.) Refers to a location you marked with the `mark` command. This is described in ``Marking Text'' later in this chapter.

You can also add a number to the end of a command to specify the number of lines involved. For example, `d5` deletes five lines, starting with the current line. If the number specified is larger than the number of lines between the current line and the end of the file, `ex` performs the operation to the end of the file.

2.2 Motion commands, paging, and scrolling

`ex` provides a number of ways to move through your file, and these are collectively referred to as motion commands. Valid `ex` motion commands are

lineno	Move to *lineno*.
`$`	Move to the last line in the buffer.
+[*n*]	Move forward from the current line. If followed by *n*, it means to move to the start of the line *n* lines forward in the buffer.
−[*n*]	Move backward from the current line. If followed by *n*, it means to move to the start of the line *n* lines backward in the buffer.
/*pattern*[/]	Move forward to a line containing the regular expression *pattern* (described under ``Regular Expressions and Searching'').
/	Move forward using the last regular expression scanned for.
//	Move forward using the last regular expression used in a substitution (described under ``Changing Text'').
?*pattern*[?]	Move backward to a line containing the regular expression *pattern* (described under ``Regular Expressions and Searching'').
?	Move backward using the last regular expression scanned for.

`??`	Move backward using the last regular expression used in a substitution (described under "Changing Text").
` `` `	(Back quote typed twice.) Move to the location before the last absolute motion.
`` `x ``	(A single back quote followed by a lowercase letter `a` through `z`.) Move to a location you marked with the `mark` command (described in "Marking Text").

In addition, you can use

CONTROL-d	Move forward half a screen. You can change this with the `set scroll` option, described under "Setting Options."
<CR>	Move to the next line.

2.3 Determining line appearance

Three commands control how text displays on the screen: `print`, `list`, and `number`.

The `print` command sets the default printing style. It displays nonprinting characters as `^`*x* and delete (octal 177) as `^?`. The format of the `print` command is

[*linespec*] `p[rint]` [*n*]

It prints the current line, the lines specified by *linespec*, or the next *n* lines. You can also add a `p` to the end of many commands to print the current line after the command completes.

The `list` command displays the specified lines with tabs indicated with `^I` and the ends of lines indicated with a `$`. The format of the `list` command is

[*linespec*] `l[ist]` [*n*]

It displays the current line, the lines specified by *linespec*, or the next *n* lines. You can also add an `l` to the end of many commands to list the current line after the command completes.

The `number` command prints the line number before each specified line. The format of the `number` command is [*linespec*] `nu[mber]` [*n*]

or

[*linespec*] # [*n*]

It prints the current line, the lines specified by *linespec*, or the next *n* lines. You can also add a # to the end of many commands to number the current line after the command completes.

See ''Setting Options'' for more information.

2.4 Determining where the current line appears

The `z` command determines where the current line appears on the screen. (If you prefix the command with a line number *lineno*, that line number becomes the current line.) There are several different forms of this command.

[*lineno*]`z`	Print the next screenful of lines with the current line at the top of the screen.
[*lineno*]`z+`	Print the next screenful of lines with the current line at the top of the screen.
[*lineno*]`z-`	Print the screen with the current line at the bottom.
[*lineno*]`z.`	Print the screen with the current line at the center.
[*lineno*]`z=`	Print the screen with the current line in the center, surrounded with lines of - characters.
[*lineno*]`z^`	Print the screen two windows before the current line.

3. Inserting text

The two basic `ex` commands for adding text to your file are the `append` and `insert` commands.

The `append` command adds text *after* the specified line. The command syntax is

[*lineno*]`a[ppend]`<CR>[*text* <CR>]`.`<CR>

where *lineno* is the line number, *text* is the text you enter, and <CR> is RETURN. To append text to the start of the buffer, use the command `0a` (this appends to line 0). A variant form, the `append!` command, changes the setting of the `autoindent` option while appending (described under "Setting Options").

The `insert` command adds text *before* the specified line. The command format is

```
[lineno]i[nsert]<CR>[text <CR>].<CR>
```

where *lineno* is the line number, *text* is the text you enter, and <CR> is RETURN. A variant form, the `insert!` command, changes the setting of the `autoindent` option during an insert (described under "Setting Options").

4. Deleting text

The `delete` command removes the specified lines. You delete a specified range of line numbers using the format

```
[linespec] d[elete] [n]
```

For example, you can delete lines 5 through 20 of your file with the command

```
5,20d
```

You can delete line 5 with the command

```
5d
```

or you can specify the line you want to delete using a regular expression. For example, if the line you want to delete ends in the word "`finish`," use the command

```
/finish$/d
```

You can also delete *n* lines with the command

```
d[elete] n
```

For example,

```
d3
```

5. Changing text

The `change` command replaces existing text with new text. The command format is [*line1*[, *line2*]]c[hange][*n*]<CR>*text*<CR>.

This changes either *line1*, the range *line1* through *line2* inclusive, or, when *n* is specified, the next *n* lines. A variant form, the `change!` command, changes the setting of the `autoindent` option during the change. (This option is described under "Setting Options.") The command format is

[*line1*[, *line2*]]c[hange]![*n*]<CR>*text*<CR>.

You can also change a string with the `substitute` command. The simplest format of this command is

s/*pattern*/*replacement*[/]<CR>

This replaces the first instance of *pattern* with the *replacement* on the current line. Regular expressions are used commonly in substitutions. For example, if you had the following line in your file:

```
At About Nine Another One Arrived
```

and you typed

```
s/A./O/
```

it would change your line to

```
O About Nine Another One Arrived
```

Substitutions are also commonly used when you want to change a word throughout your file. The common format for this is

`1,$ s/`*pattern/replacement/*`g`

This tells `ex` to make the replacement on every line of the file (`1,$` means from the first line to the last line); `g` tells it to make the replacement every time it appears in a line. If you didn't add the `g`, it would make the replacement only once per line, at the first appearance of *pattern*.

A variation of this format is useful when you want to repeat the same change several times within one line. Instead of giving `1,$` for the range of the command, you may specify a single line. For example, if the above sample line were the current line (or dot, "`.`") in your file, you could type

```
s/A./O/g
```

to change your line to

```
O Oout Nine Oother One Orived
```

In general, `substitute` replaces the first instance of *pattern* with *replacement* on each specified line. The *suffixes* are

`g` (global.) Substitute *pattern* with *replacement* every time it appears on the specified lines. To make the substitution everywhere in the file, use the format

`1,$s/`*pattern*`/`*replacement*`/g`

You can also use `%` instead of `1,$`.

`c` (confirm.) Print the line before making each substitution, marking the string to substitute with ^ characters. Type `y` to confirm the substitution, and type any other character if you don't want to make the substitution.

`r` (replace.) Replace the previous replacement pattern from a substitution with the most recent search string.

You can split lines by substituting newline characters into them. You must escape the newline in *replacement* by preceding it with a backslash (\). See "Regular Expressions and Searching" for other metacharacters available in *pattern* and *replacement*.

Omitting *pattern* and *replacement* repeats the last substitution. For example, if you subsitute the word `test2` for the word `test` on one line using the command

```
s/test/test2/
```

you can repeat this substitution on another line by typing

```
s
```

This is a synonym for the `&` command, which is described later in this chapter.

Using the `r` suffix (`sr`) replaces the previous *pattern* with the previous regular expression. For example, if you make the substitution

```
s/test/test2
```

then search for a *pattern* such as `my.test`,

```
/my.test/
```

Then the command

```
sr
```

Changes `my.test` to `test2`. If you omit the `r` suffix, the `s` command replaces `my.test` with `my.test2`. This is a synonym for the ~ command, which is described later in this chapter. See ‘‘Command Summary’’ and ‘‘Repeating the Last Command’’ in Chapter 4, ‘‘Using `vi`,’’ for more information on repeating substitutions.

6. Copying and moving text

You can use several commands to copy and move text.

The `copy` command places a copy of one section of your text after the specified line. The most common format of this command is

line1`,` *line2*`co[py]` *lineno*

This copies the lines between *line1* and *line2* and places them after *lineno.*

The `move` command moves a section of your text to a new location in your file. The format of this command is

`[`*line1*`[,` *line2*`]]m[ove]` *lineno*

This moves the text either from *line1* or between *line1* and *line2* after *lineno.*

You can also move text by moving it into `ex`’s buffer and then placing it in the text with the `put` command.

There are two general forms of commands used for moving text in this way. The first is to use the `yank` or `delete` commands to place the text in `ex`’s unnamed buffer. This happens automatically when you use the `delete` command to delete the text (described under ‘‘Deleting Text’’), or the `yank` command to copy the text. The general forms for the `yank` command are

```
[line1[, line2]]ya[nk]
```

to place a copy of the text either from *line1* or between *line1* and *line2* in `ex`'s unnamed buffer, or

```
ya[nk] n
```

to place a copy of the next *n* lines in `ex`'s unnamed buffer. You can then place `ex`'s buffer somewhere else in the file with the `put` command (either by moving to where you want the text to appear, or by specifying an address before the `put` command). If you use `ex`'s unnamed buffer, you can't make any modifications to your text between placing the text in your buffer and putting it in its new location.

The other way to move or copy text is to use one of `ex`'s named buffers. `ex` has 26 buffers named `a` through `z`. Use the same general format as before, but specify a buffer name with each command. For example,

```
1,4 d a
```

deletes lines 1 through 4 and places them in buffer `a`.

```
10 pu a
```

puts the contents of buffer `a` after line 10.

You can also specify the buffers as `A` through `Z` if you want the text you are currently deleting or yanking to be appended to the end of the buffer rather than overwriting it.

You can use `ex`'s named buffers to move information from one file to another if you have specified both files when you started `ex`.

7. Regular expressions and searching

A **regular expression** uses **metacharacters** (special characters that stand for other characters) to stand for a set of strings. The regular expression is said to match each element in this set of strings. For example, if `.` is a special character standing for any letter and `A` is an ordinary character standing for `A`, then `A.` would find all of the following words: `At, About, Another.`

Regular expressions in `ex` always appear between the characters `/ /` or the characters `? ?`.

The following characters appear in regular expressions:

char	Any character other than the metacharacters listed below matches itself. The characters listed below are metacharacters. You have to precede them with a backslash (\) to have `ex` treat them as ordinary characters.
^	At the beginning of a pattern, the caret specifies that the pattern is at the beginning of a line. For example, `^A` specifies a line beginning with `A`. This character has a different meaning within square brackets.
$	At the end of a pattern, the dollar sign specifies that the pattern is at the end of a line. For example, `a$` specifies a line ending with `a`.
.	The period matches any single character except the newline character. For example, `A.` matches `A` followed by any character.
[*pattern*]	A pattern enclosed in square brackets sequentially matches a set of single characters defined by *pattern*. • Ordinary characters in *pattern* match themselves. For example, `[ab]` specifies either `a` or `b`. • A pair of ordinary characters separated by – in *pattern* defines a range of characters. For example `[a-z]` matches any lowercase letter. • If ^ is the first character within square brackets, it specifies characters that are not in the pattern. For example, `[^a-z]` matches anything but a lowercase letter.
\<	This matches a pattern at the start of a word (`ex` defines the start of a word as the beginning of a line, or a letter, digit, or underline that follows any character other than a letter, digit, or underline).
\>	This matches a pattern at the end of a word (`ex` defines the end of a word as the end of a line, or a letter digit or underline followed by any character other than a letter,

digit, or underline).

You can use the preceding regular expressions to construct larger regular expressions using the following rules:

- If you use two regular expressions (for example, `[a-z][A-Z]`), the editor matches the first string it encounters that matches both regular expressions in the order they appear.
- Following a regular expression with an asterisk (`*`) matches *zero or more occurrences* of the preceding character. Generally, you should use this within a longer regular expression, since it matches for zero occurrences first. That is, searching for `a*` finds zero occurrences and matches the characters following the cursor. This is convenient, however, for longer regular expressions. For example, `ba*b` matches `bb`, `bab`, `baab`, `baaab`, and so on. Within a longer regular expression, if there is any choice, it matches the longest leftmost string. In the preceding example, it would choose `baaab`.
- Enclosing a regular expression in `\(` and `\)` defines the regular expression as a numbered "field," so that you can refer to it later. For example,

  ```
  \([a-z]\)\([A-Z]\)
  ```

  defines two fields, numbered sequentially. Use `\`*f* (where *f* is the number of the field) to refer to any of the fields in the regular expression. In the previous example `\1` refers to the field matched by the regular expression `[a-z]` and `\2` refers to the field matched by the regular expression `[A-Z]`.
- The null regular expression (`//` or `??`) is shorthand for the last regular expression.

7.1 Turning off metacharacters

There are two ways to use `ex`'s metacharacters as ordinary characters (if, for example, you want to search for the character `.`).

1. You can precede the metacharacter with a backslash (`\`).
2. You can set the `nomagic` option (see "Setting Options"). This strips all but the following three metacharacters of their special meaning: `^` at the beginning of a regular expression (indicating

the beginning of a line), `$` at the end of a regular expression (indicating the end of a line), and the backslash character (\). You can restore the special meaning to the other metacharacters by preceding them with a backslash (\).

8. Working with multiple files

You can work with several files during one editing session. Before describing the commands for this, we define the terms that refer to these files.

8.1 The current file

The file you are editing is the **current file.** This means that the buffer contains the edited version of this file. `ex` overwrites this file with the updated version in the buffer without protest. When `ex`'s current file is not the file being modified in the buffer (for example, if you use the `file` command to change the name of the current file without changing the buffer)

```
f filename
```

`ex` does not overwrite a file with the buffer's contents. See ''Changing the Current File.'' If the file in the buffer is not the current file, the `file` command displays

```
[Not Edited]
```

You can use `%` to refer to the current file anywhere you would use the filename.

8.2 The alternate file

`ex` also has an **alternate file**, which is usually the previous file you edited.

If you haven't previously edited another file, but have read a file into the buffer with the `read` command, this becomes the alternate file.

You can use # to refer to the alternate file anywhere you would use the filename. This makes it easy to alternate between two files. For example, if you are in the current file and want to edit the alternate file, type

```
e #
```

This reads the alternate file into the buffer, making it the current file. (If you had modified the buffer since you last wrote it to file, `ex` would warn you and would not edit the alternate file.)

8.3 Specifying multiple files at startup

You can specify more than one file when you start `ex`. The format is

```
ex filename ...
```

`ex` reads the first file into the buffer and creates an **argument list** containing the names of all the files you specified.

8.3.1 Displaying the argument list

The `args` command displays the current argument list with the current file delimited by brackets (`[ ]`).

8.3.2 Editing the next file on the argument list

The `next` command edits the next file in the argument list. The format for this command is

```
n[ext]
```

You must save any changes you have made before editing a new file or you'll get the message

```
No write since last change (:next! overrides)
```

To edit the next file in the argument list and overwrite the current buffer with this file, type

```
n!
```

The buffer is overwritten with the next file, whether you've saved the current buffer or not.

Typing

```
n+cmd
```

executes *cmd* after opening the first file on the argument list.

8.3.3 Replacing the argument list

You can replace the list of files in the argument list with another list of files by typing

```
n filename ...
```

ex replaces the list of files in the argument list and edits the first file on the new list.

If you made changes to one or more files on the original argument list and you haven't saved the current buffer, you'll get the message

```
   No write since last change (:n! overrides)
```

If you haven't saved the buffer, you can force ex to replace the argument list with the new list by typing

   n! *filename* ...

The new list of files replaces the original one even if you have not saved the current buffer.

8.3.4 Returning to the first file on the argument list

The rewind command edits the files on the argument list beginning with the first file. The format of the command is

```
   rew
```

If you made changes to one or more files on the original argument list and you haven't saved the current buffer, you'll get the message

```
   No write since last change (:rewind! overrides)
```

Typing

```
   rew!
```

forces ex to edit the files on the argument list beginning with the first file and discarding any changes you made to the current buffer.

8.4 Editing a new file

The edit command reads a new file into the buffer. If you haven't saved the current buffer, ex warns you and doesn't edit the new file. If you have saved the current buffer, ex deletes the buffer contents, makes the specified file the current file, and prints the new filename and the number of lines and characters read.

ex sets the current line to the last line in the new file (in line mode) or the first line in the new file (in open or visual mode).

ex strips the high-order bit from any non-ASCII characters and discards any null characters.

If the file is a special device (block, character, or TTY), `ex` tells you and allows you to edit the file. If it is a directory, you'll get the message "`Directory.`" If it is a binary file, you'll get the message "`Line too long`" or "`Incomplete last line.`" Generally, you shouldn't edit these kinds of files.

The `edit!` command edits the specified file and overwrites the current buffer, whether you've written it or not. The form of this command is

`e!` *filename*

Typing

`e+`*lineno filename*

or

`e+/`*pattern filename*

begins editing *filename* at line *lineno* or at pattern /*pattern* (*pattern* can't contain spaces).

8.5 Copying another file to the current buffer

The `read` command copies the text of *filename* to the current buffer after the specified line. The format is

[*lineno*]`r[ead]` [*filename*]

where *lineno* is a line number, or an expression resulting in one. If you don't specify a *filename*, it uses the current filename. If there is no current filename, *filename* becomes the current name. If the file buffer is empty and there is no current file, `ex` treats this as an `edit` command.

`read` then tells you the filename read in and the number of lines and characters read. After a `read` command, the current line is the last line read (in `ex`) or the first line read (in open or visual mode).

Typing

```
0read
```

reads the file at the beginning of the buffer.

8.5.1 Finding out more about the file you're in

The `file` command tells about the file you are editing. Its syntax is

```
f[ile]
```

It prints the following information:

- the current filename
- whether you have modified the current file since the last `write`
- whether the current file is in read-only mode
- the current line
- the number of lines in the buffer
- the current line's position in the buffer, given as a percentage from the beginning of the buffer

It also notes when the current file is ''not edited'' (the current file is not the file in the buffer). In this case, you have to use `w!` to write to the file, since `ex` does not know if `write` will destroy a file unrelated to the current buffer contents.

8.5.2 Changing the current file

The `file` command changes the current filename to *filename* without changing the buffer contents. The format of the command is

`f[ile]` *filename*

The current file is then considered ''not edited.''

9. Using shell commands in ex

There are several ways of interacting with the shell from within `ex`.

9.1 Running another program from ex

The `!` character invokes a shell to execute a single command using the syntax `!`*command*. This executes *command* in the shell and returns you to the editor when the command completes.

The command you invoke from the editor using the `!` syntax may be a simple command such as `ls`, or it may be an interactive program such as `dc` or a shell script.

If you enter a simple shell command from the editor, it executes immediately and prints `!` on the screen when it terminates. You are then back in the editor at the same position in the file. For example,

```
!ls
```

lists the files in your current directory. If you haven't written the buffer contents since the last change, `ex` prints a warning message before executing the command. Before returning you to the editor, it prints "`!`".

If you enter an interactive program from the editor, it runs until you exit that program. For example,

```
!dc
```

invokes the `dc` calculator program. You can then use `dc` as long as you wish. When you terminate `dc`, you are back in the editor at the same position in the file.

The command

```
sh
```

invokes your login shell. You may then give as many commands in the shell as you wish. When you finish with the shell, type an *eof* character (CONTROL-d in the A/UX standard distribution) or

```
exit
```

to return to the editor.

You may escape to a shell different from your login shell. The general form for this command is

!*shell*

where *shell* is the name of the shell you wish to invoke. For instance, if your login shell is the Bourne shell (`sh`(1)), you may invoke the `csh` instead with the following:

```
!csh
```

This invokes a copy of the C shell, temporarily suspending `ex`. You may then give as many commands in the new shell as you wish. When you finish with the shell, type an *eof* character (CONTROL-d in the

A/UX standard distribution) or

```
exit
```

to return to the editor.

Remember that after you use the `sh` (or `!`*shell*) escape from `ex`, you have invoked a new shell, not the shell from which you initiated `ex`. If you use `sh` to escape `ex` and forget that you have suspended your editing job, you might invoke a new copy of `ex` from your new shell instead of exiting that shell and going back to the original `ex` session. This can cause problems with inconsistent versions of a file if you finally quit `ex`, and is something to be aware of when using `sh` or `!`*shell* from `ex`.

9.2 Directing command output to the buffer

You can read the output of a command into your file with the `read !` command. The usual format of this command is

`r[ead] !`*command*

(you must type a blank or tab before the `!`). For example,

```
read !ls
```

places the directory listing after the current line.

Note: The shell prompts are also written to the file.

9.3 Sending the buffer to shell commands

You can send part of your buffer to a command with the `write !` command (you must precede the `!` with a blank or a space). The format of this command is

[*line2*[, *line2*]]`w[rite] !`*command*

For example, to format the first 20 lines of your file without leaving the editor, use the command:

```
1,20 w !nroff > new.file
```

When you precede this command by a range of line numbers, `ex` sends the specified line(s) to *command* and replaces the line(s) with the output of the command.

9.4 Writing shell scripts using ex commands

You can write shell scripts that use `ex` commands. You add comments to these scripts by starting a line with a double quote (") or by adding a double quote and a comment to the end of a command (except when they could be read as part of the command—as in shell escapes and the `substitute` and `map` commands).

You can place more than one command on a line by separating each pair of commands with a | character. If you use a global command, comment, or shell escape (`!`), however, it must be the last command on the line.

You can also write multiple-line commands by ending each line except the last line with a backslash (\).

You can use the – flag option to `ex` within shell scripts to suppress interactive feedback. This permits the script to run without pausing for information to be typed in from the terminal.

The following is an example of a shell script named `script.ex`.

```
for i in $*
do
ex - $i <<EOF
g/\\f1/s//\\fR/g  "change \f1 to \fR
g/\\f2/s//\\fI/g  "change \f2 to \fI
g/\\f3/s//\\fB/g  "change \f3 to \fB
wq
EOF
done
```

To run `script.ex` on a text file, make the script file executable with the command

```
chmod +x script.ex
```

Then type

`script.ex` *filename*

where *filename* is a text file. When the script has finished running, all instances of ``\f1'' in your file are changed to ``\fR,'' and so on. See also Chapter 6, ``Using `sed`'' for information on making global changes to a file using scripts.

10. Setting options

You control many of the ways `ex` behaves by setting options.

`ex` has three kinds of options: numeric, string, and toggle. Each of these options is set in its own way. **Numeric options** are options that take a numeric value and **string options** are options that take a string value. You set numeric and string options with the following command format:

```
set opt=val
```

For example, you set the number of lines to scroll through (a numeric option) with the following command:

```
set scroll=4
```

You set the default shell (a string option), with the following command:

```
set sh=/bin/sh
```

A **toggle option** is an option that is either on or off. You set a toggle option with one of the following formats:

```
set opt
```

turns the option on and

```
set noopt
```

turns the option off. For example,

```
set number
```

turns on line numbering and

```
set nonumber
```

turns off line numbering.

To set options, use the following syntax:

Option type	Syntax	Sets *opt* to
numeric	`set` *opt=number*	*number*
string	`set` *opt=string*	*string*
toggle	`set` *opt*	on
	`set no`*opt*	off

You can place multiple options on one line with the format

```
set  opt opt opt
```

Most options can be abbreviated; see ''Option Summary'' for a list of options and their abbreviations.

10.1 Listing options

To see option settings, use the following syntax:

Options listed	Syntax
all	`set all`
changed ones	`set`
an *opt*	`set` *opt*?

10.2 When to set options

You can set these options anytime while editing a file, or you can set them as part of your default editing environment by including them in the `EXINIT` variable in your `.profile` file or by creating a `.exrc` file in the current or home directory. If you set them in your `.profile` file, they should all be on one line, separated by vertical bar (|) characters. For example, in your `.profile` file, you could have the following:

```
EXINIT="set number|set scroll=20|set terse"
```

10.3 Option descriptions

The following is a complete list of available options:

`autoindent`
Abbreviation: `ai`
Default: `noautoindent`

Begin new lines of text at the indent level of the previous line. This is useful in structured program text.

When inserting text, CONTROL-d moves the cursor back to the previous tab stop.

Entering a blank line or typing `0`CONTROL-d erases the autoindent. You can type one line at the margin by beginning it with ^CONTROL-d; the next line returns to the previous indent.

`autoindent` does not work with global commands or when the input device is not a terminal.

`autoprint`
Abbreviation: `ap`
Default: `autoprint`

Print the current line after each `delete`, `copy` (or `t`), `join`, `move`, `s` (substitute), `undo`, <, or > command (if it is the last command on the line). This is suppressed during global commands.

`autowrite`
Abbreviation: `aw`
Default: `noautowrite`

Write the modified buffer contents to the current file when you use the following `ex` commands:

`n[ext]`	edit next file in series
`rew[ind]`	reedit the list of files from the start
`ta[g]`	move to a tag location
`!`	escape to the shell

or the following `vi` commands:

CONTROL-^	switch files
CONTROL-]	move to a tag location

You can override the `autowrite` option, destroying the current buffer contents, with the following `ex` commands:

`e[dit]`	instead of `n[ext]`
`rew[ind]!`	instead of `rew[ind]`
`ta[g]!`	instead of `ta[g]`
`sh[ell]`	instead of `!`

and the following `vi` commands:

`:e #`	instead of CONTROL-^
`:ta!`	instead of CONTROL-]

`beautify`
Abbreviation: `bf`
Default: `nobeautify`

Discard all control characters (except tab, newline, and form feed) from your input, and print a message the first time it discards a backspace character. This option applies only to text input and not to command input.

`directory=`*dir*
Abbreviation: `dir`
Default: `directory=/tmp`

Specify the directory where `ex` places its buffer file. This directory must be writable, or `ex` will exit abruptly.

`edcompatible`
Abbreviation: none
Default: `noedcompatible`

Use the suffixes `g` and `c` to toggle globally and confirm settings of the `s` (substitute) command.

`errorbells`
Abbreviation: `eb`
Default: `noerrorbells`

Sound a bell when displaying error messages (you cannot suppress this bell in open or visual mode). If possible, `ex` highlights the error message on the screen instead of ringing the bell.

`flash`
Abbreviation: `fl`
Default: `flash`

Flash the screen when an error occurs.

`hardtabs=`*n*
Abbreviation: `ht`
Default: `hardtabs=8`

Set the length of terminal hardware tabs (or where the system expands tabs).

`ignorecase`
Abbreviation: `ic`
Default: `noignorecase`

Set regular expressions to match both uppercase and lowercase patterns, except when you specify a range of uppercase characters (for example, `[A-Z]`).

`insertarrows`
Abbreviation: `ia`
Default: `insertarrows`

Allow use of arrow keys in insert mode as well as command mode.

`lisp`
Abbreviation: none
Default: `nolisp`

Set `autoindent` and modify the `vi` motion commands `()`, `{}`, `[[`, and `]]` to have meaning for LISP.

`list`
Abbreviation: `list`
Default: `nolist`

Print lines showing tabs as `^I` and the end of the line as `$`, as in the `list` command.

`magic`
Abbreviation: none
Default: `magic`

With `magic` set, `ex` recognizes all the metacharacters used in regular expressions. Setting `nomagic` uses only the following regular-expression metacharacters: \, ^, and `$`. It treats all other characters (including ~ and & used in substitutions) as normal characters. You can use any of these as metacharacters by preceding them with \.

`mesg`
Abbreviation: none
Default: `mesg`

By default, `vi` allows other users to send you messages while you are editing a file. `nomesg` turns off this permission.

`number`
Abbreviation: `nu`
Default: `nonumber`

Print lines preceded by their line number and (after RETURN) prompt with line numbers for input lines.

`open`
Abbreviation: none
Default: `open`.

> By default, you can enter `open` and `visual` mode from `ex`. Setting `noopen` means you can't use these modes.

`optimize`
Abbreviation: `opt`
Default: `optimize`

> Suppress carriage returns on more than one (logical) line of output. This optimizes output on terminals without addressable cursors when printing text with leading white space.

`paragraphs=`*xyz*
Abbreviation: `para`
Default: `paragraphs=PLIbp`

> Specify the paragraph macro searched for in `vi` and `open` mode when you type `{` or `}`. By default, it searches for the `mm` macros `.P`, `.LI`, and the `nroff/troff` request `.bp`.

`prompt`
Abbreviation: none
Default: `prompt`

> By default, `ex` prints the prompt (`:`) when it is in command mode. Setting `noprompt` turns off this prompt.

`readonly`
Abbreviation: none
Default: `noreadonly`

> Make the file read only (just as if you had started `ex` with the `-r` flag option). You can override this option and save the file by using the `write!` command.

`redraw`
Abbreviation: none
Default: `noredraw`

> Force a dumb terminal to redraw the characters to the right of the cursor as you type input in `vi`. This is useful only at baud rates of

1200 or higher.

`remap`
Abbreviation: none
Default: `remap`

Repeatedly translate maps until they are unchanged. For example, if you map o to O, and O to I, setting `remap` maps o to I, while setting `noremap` maps o to O.

`report=`*n*
Abbreviation: none
Default: `report=5`

Print a message when a command modifies more than *n* lines. For example, `ex` prints `12 lines deleted` after a deletion. For the following commands, `ex` reports after completing the entire command: `global`, `open`, `undo`, and `visual`.

`scroll=`*n*
Abbreviation: none
Default: `scroll=`half the value of the `window` option

Set how many lines scroll when you press CONTROL-d in `vi`'s command mode. By default, it uses half the number of lines set with the `window` option.

`sections=`*xyz*
Abbreviation: none
Default: `sections=HHU`

Specify the section macro searched for in `visual` and `open` mode when you type `[[` or `]]`. By default, it searches for mm's `.H` and `.HU` macros.

`shell=`*pathname*
Abbreviation: `sh`
Default: `shell=$SHELL`

Set the pathname of the shell used by the shell escape command `!` and the `shell` command. `$SHELL` is the value in the `SHELL` variable, as set in the `.login` or `.profile` file.

`shiftwidth=`*n*
Abbreviation: `sw`

Default: `shiftwidth=8`

> Set the software tab stop width used when you press CONTROL-d in `autoindent`, and when you use the > and < commands.

`showmatch`
Abbreviation: `sm`
Default: `noshowmatch`

> Move the cursor to the matching (or { on the screen for one second when you type `(` or `{` in `vi`. Extremely useful with LISP.

`slowopen`
Abbreviation: `slow`
Default: `noslow`

> Don't update the display when you enter text in `vi`. Useful on a very slow line.

`tabstop=`*n*
Abbreviation: `ts`
Default: `tabstop=8`

> Set tabstops to *n*.

`taglength=`*n*
Abbreviation: `tl`
Default: `taglength=0`

> Tags are not significant beyond *n* characters. Zero (the default) means that all characters are significant.

`tags=`*pathname*
Abbreviation: none
Default `tags=/usr/lib/tags`

> Search for the requested files sequentially in the specified path when using the `tag` command. You must escape any spaces with a backslash (\). By default, it searches in the current directory and in `/usr/lib` (a master file for the entire system).

`term=`*terminal*
Abbreviation: none
Default: `$TERM`

Set your terminal type. The value you specify must exist in a file in the appropriate subdirectory of `/usr/lib/terminfo`. `$TERM` is the value of the `TERM` variable, as set in the `.login` or `.profile` file.

`terse`
Abbreviation: none
Default: `noterse`

Produce shorter error messages.

`warn`
Abbreviation: none
Default: `warn`

Print `[No write since last change]` if you use a `!`*command* escape before you have saved the current buffer.

`window=`*n*
Abbreviation: None
Default: speed dependent

Set the number of lines in `vi`'s text window. By default, this is determined by your baud rate. The default settings are

`8`	for 600 baud or lower
`16`	for 1200 baud
`23`	(or full screen) at higher speeds (w300, w1200, w9600).

These settings set `window` only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an `EXINIT` variable.

`wrapmargin=`*n*
Abbreviation: wm
Default: `wrapmargin=0`

Set the column number where the cursor automatically returns when you enter text in visual and open modes. This is determined by setting the wraparound point *n* columns from the right side of the screen. Since there are 80 columns on a typical terminal screen, setting `wrapmargin=10` would break a long line 10 columns to the right of this, in the 70th column. `wrapmargin=0` means that a long line wraps at column 80, but is not broken (that is, `wrapmargin` is off).

`wrapscan`
Abbreviation: `ws`
Default: `wrapscan`

Search the entire file for a regular expression by moving from the current line, wrapping around the end or beginning of the file (depending on the direction you're searching in), and returning to the current line.

`writeany`
Abbreviation: `wa`
Default: `nowriteany`

By default, `ex` warns you if you try to save your buffer to any file other than the current file. Setting `nowriteany` allows you to write to any file with write permission.

10.4 Option summary

The following is a complete list of `ex` editor options.

Option	Abbreviation	Default
`autoindent`	`ai`	`noai`
`autoprint`	`ap`	`ap`
`autowrite`	`aw`	`noaw`
`beautify`	`bf`	`nobf`
`directory`	`dir`	`dir=/tmp`
`edcompatible`	—	`noedcompatible`
`errorbells`	`eb`	`noeb`
`hardtabs`	`ht`	`ht=8`
`ignorecase`	`ic`	`noic`
`lisp`	—	`nolisp`
`list`	—	`nolist`
`magic`	—	`magic`
`mesg`	—	`mesg`
`number`	`nu`	`nonu`

Option	Abbreviation	Default
open	—	open
optimize	opt	opt
paragraphs	para	para=PLIbp
prompt	—	prompt
readonly	—	noreadonly
redraw	—	noredraw
remap	—	remap
report	—	report=5
scroll	—	scroll=1/2 window
sections	—	sections=HHU
shell	sh	sh=$SHELL
shiftwidth	sw	sw=8
showmatch	sm	nosm
slowopen	slow	(terminal dependent)
tabstop	ts	ts=8
taglength	tl	tl=0
tags	—	tags=/usr/lib/tags
term	—	term=$TERM
terse	—	noterse
warn	—	warn
window	—	window=(speed dependent)
wrapmargin	wm	wm=0
wrapscan	ws	ws
writeany	wa	nowa

11. Mapping and abbreviations

Mapping and abbreviation are available as a joint facility of ex and vi. The mapping or abbreviation must be defined on the ex command line, *but is useful only in visual mode*. See ''Mapping and Abbreviations'' in Chapter 4, ''Using vi.''

12. Other ex commands

This section describes several other ex commands. See ''Command Summary'' for a complete list of ex commands and their usage.

12.1 Marking text

You can set up a special address with the `mark` command and then use this address anywhere you would use an `ex` address. The syntax for this command is

```
[lineno]ma[rk] x
```

where *lineno* is the line (number) in the file to mark and *x* is a lowercase letter to mark this line. You must precede this letter with a blank or a tab. After you have marked a line, you can refer to it by typing

```
'x
```

where *x* is the name you gave it.

You can also use the `k` command to mark a line. The syntax for this command is

```
[lineno]kx
```

where *lineno* is the line (number) in the file to mark and *x* is a lowercase letter to mark this line. This is the same as the `mark` command, except you don't have to precede the letter with a space.

12.2 Recovering lost text

There are several ways to recover information in `ex`.

The `undo` command changes the buffer back to the way it was before the last editing command.

If you have deleted several large sections and want to recover them, the easiest method is to exit `ex` without saving your changes by using the `quit!` command. The format for this command is

```
q[uit]!
```

This restores the buffer you had when you last wrote the file.

The `preserve` command is a more drastic way to save your file. You should use it only if you can't save your file with a `write` command. It saves your file as though your system had just crashed. You can recover this file with the `recover` command.

12.3 Editing programs

ex has several commands for editing programs.

< shifts text right to the next tabstop and > shifts text left. This is useful for changing the indentation of a section of program.

Using the tag command helps you locate functions that may be spread over many files. The tag command starts editing the file at *tag*, moving to another file, if necessary. Since the current file edit may be terminated abruptly, you must write the current file, if you modified it, before giving a tag command. If you give the tag command without specifying a *tag*, ex uses the previous tag. The syntax is

```
ta[g] tag
```

Normally, you use this command after using the ctags command to create a tag file (see ctags(1) in *A/UX Command Reference*). This file consists of several lines with three fields separated by blanks or tabs. The first field is the name of the tag, the second is the name of the file where the tag resides, and the third is an addressing form used to find the tag. Usually, this is a contextual scan in the form */pattern/*, performed as if nomagic were set.

Names in the tag file must be sorted alphabetically.

For instance, if you wish to have ready access to the functions in a file called functions.c of the following form,

```
main()
{
                f1();
                f2();
                f3();
}
 f1()
{
                f2();
}
```

```
 f2()
{
                f3();
}
 f3()
{

}
```

you could run the `ctags` program on it, as follows:

```
ctags functions.c
```

This creates a tag file called `tags`.

If you then edit `functions.c` *or even another file* with `ex`, and give the command

```
ta f1
```

you will find yourself in the editor buffer of `functions.c` at the line

```
 f1()
```

ready to edit that function.

> *Note:* If you had been editing a file other than `functions.c` when you gave the `tag` command, that other file's edit would have been dropped summarily without a `write`. Always `write` *before* giving a `tag` command.

13. Saving text and exiting

The `write` command saves the changes you've made to the buffer in the specified file and prints the number of lines and characters written. The format of this command is

[*line1*[, *line2*]]`w[rite]` [*filename*]

By default, `ex` writes the entire buffer to *filename*. Including a starting and ending address saves only the lines between *line1* and *line2*. Including just a starting address saves only *line1*. If you don't include *filename*, ex writes to the current file. If there is no current file, ex creates the file *filename* and writes the buffer to it. The write command also writes to /dev/tty and /dev/null.

The write! command forces ex to write the buffer. The file must already exist. The format of this command is

[*line1*[, *line2*]]w[rite!] [*filename*]

The write>> *filename* command appends the buffer to the end of a file (which must already exist). The format of this command is

[*line1*[, *line2*]]w[rite] >> *filename*

13.1 Exiting ex

The quit command terminates ex using the syntax

```
q[uit]<CR>
```

If you haven't written all your changes to a file, ex warns you and does not quit. It also tells you if there are additional files in the argument list.

The quit! command allows you to leave ex without saving the changes you've made, using the syntax

```
q[uit]!<CR>
```

See "Saving Text and Exiting" for how to write a file.

13.2 Combining the write and quit commands

The following commands save your changes and exit ex.

The write and quit commands can be used together to save your changes in the current file or in a specified file and then exit ex.

```
wq
```

This is simply a shorthand form of using write followed by quit; you may also use the long form of these commands as shown in the sections above.

Followed by an exclamation point,

wq! *filename*

forces ex to save your changes in *filename* and then exits ex. This is a shorthand form of using `write!` followed by `quit`. You may also use the long form of these commands as shown in the sections above.

The `xit` command saves your buffer only if you have made changes since last saving it and then exits `ex`. You can use this command by typing

```
x
```

The long format of this command is

`x[it]` [*filename*]

where *filename* is the name of the file in which to save your buffer.

14. Error conditions

When there is an error, `ex` sounds the terminal bell and prints an error message. If `ex` receives an interrupt signal, it prints `Interrupt` and returns to command mode. If the primary input is from a file, `ex` exits.

14.1 Limitations

`ex`'s limitations are

- 1024 characters per line
- 256 characters per global command list
- 128 characters per filename
- 128 characters in the previous inserted and deleted text in open or visual mode
- 100 characters in a shell escape command
- 63 characters in a string option
- 30 characters in a tag name

A limit of 250,000 lines in the file is silently enforced.

The number of macros defined with `map` in `vi` is limited to 32, and the total number of characters in macros is limited to fewer than 512.

14.2 Recovering lost files

If the system crashes or you accidentally hang up before saving your file, `ex` sends you mail informing you it has saved your file. The `-r` flag option recovers your buffer. The format of this command is

```
ex -r
```

to list the files that `ex` saved, and

`ex -r` *filename*

to recover *filename* (after first moving to the directory you were in). You should check the recovered file before overwriting the existing file with it. Note that you must use the `write` command to explicitly write the recovered file. The `xit` command is a safe way to exit a recovered file because it does not guarantee that a write operation will occur.

15. Command summary

In the following commands, the last line you enter, copy, change, or print becomes the current line. If you use an input command, but don't enter a new line, the next line becomes the current line. If you delete text, the following line become the current line; deleting text at the end of the file makes the new last line the current line.

The following is the standard `ex` command format:

[*line-spec*] *command*[!] [*parameters*] [*n*] [*flags*]

Pressing only RETURN prints the next line. `ex` ignores a `.` preceding any command.

In these commands,

line-spec	(for "line-specifier") indicates the command address (see "Line Selection" for the definition of *line-spec*). In the following commands, *line-spec* defaults to the current line, unless stated otherwise.
lineno	(for "line number") indicates the single-line command address (defaulting to the current line).
flags	indicates invoking the printing command # (`number`), `l` (`list`), or `p` (`print`) after the command.
command n	performs *command* on the *n* lines involved, starting at the current line.

Command	Description
`ab[breviate]` *wd word*	(Must be defined in `ex`, but works in `vi` only.) Abbreviate *word* to *wd* in input mode. Typing *wd* in input mode, delimited by spaces or punctuation, displays *word.*
[*lineno*]`a[ppend]`<CR>*text*<CR>`.`	Append *text* after *lineno*. Specify *lineno* zero (0) to insert text at the beginning of the buffer. Default address: current line.
[*lineno*]`a[ppend]!`<CR>*text*<CR>`.`	Same as `append`, but changes the setting for `autoindent` while appending (see ‘‘Setting Options’’). Default address: current line
`ar[gs]`	Print the list of files to edit, with the current file delimited by brackets (`[ ]`).
[*line1*[`,` *line2*]]`c[hange]`[*n*]<CR>*text*<CR>`.`	Replace *line1*, or the lines between *line1* and *line2*, or *n* lines, with *text*. Default address: current line.
[*line1*[`,` *line2*]]`c[hange]!` [*n*]<CR>*text*<CR>`.`	Changes the setting of the `autoindent` option while changing the text (see ‘‘Setting Options’’). Default address: current line.

Command	Description
[*line1*[, *line2*]]co[py] *lineno* [*flags*]	Copy the text on *line1*, or between *line1* and *line2*, after *lineno* (if *lineno* is zero (0), the text is copied to the start of the file). Including a *flag* after the command (either p for print, l for list, or # for number) changes the display to the specified format. t is a synonym for copy. Default address: current line.
[*line1*[, *line2*]]d[elete] [*buffer*] [*n*] [*flags*]	Delete the specified lines (either *line1*, those lines between *line1* and *line2*, or the number of lines specified with *n*). Specifying *buffer* with a lowercase letter (a through z), overwrites that buffer with the deleted text, while specifying an uppercase letter (A through Z) appends the deleted text to that buffer. Including a *flag* after the command (either p for print, l for list, or # for number) changes the display to the specified format. Default address: current line.
e[dit] *file*	Edit *file*. ex warns you if you haven't written your current buffer, and doesn't allow you to edit *file*. Otherwise, ex reads *file* into the buffer, making it the current file, and prints the new filename and the number of lines and characters read. ex strips the high-order bit from any non-ASCII characters and discards any null characters.

Command	Description
	The current line is the last line (in `ex`), or the first line (in `open` or `visual` mode). If *file* is a special device (block, character, or TTY), `ex` mentions this, but allows you to edit the file. If it is not a text file, it prints the error message `Line too long` (or `Directory`, if appropriate).
`e[dit]!` *file*	Same as `edit`, but `ex` doesn't warn you if you haven't saved the current buffer.
`e[dit]+`*lineno file*	Begin editing *file* at line *lineno*, where *lineno* may be a line number or a pattern */pattern* (*pattern* can't contain spaces).
`f[ile]`	Print the following information: current filename; whether you have saved the current buffer; whether the current file is in read-only mode; the current line; the number of lines in the buffer; the percentage of the way through the buffer of the current line. It also notes when the current file is not the file in the buffer, by printing `not edited`. This happens if you change the current file with the `file` *file* format. In this case, you have to use `w!` to write to the current file.
`f[ile]`*file*	Change the current filename to *file*, without changing the buffer. *file* is then considered "not edited."

Command	Description
[*line1*[, *line2*]]g[lobal]/*pat*[/*cmds*/]	global/*pat* prints lines containing the regular expression *pat* and global/*pat*/*cmds*/ performs *cmds* at lines containing the regular expression *pat*. *cmds* can span several lines if you end all but the last line with a backslash (\). *cmds* can include append, insert, or change. If one of these is the last command, you can omit the period that terminates these commands. *cmds* can also include the open or visual commands, which take input from the terminal. *cmds* cannot include global or undo, since undo would reverse the entire global command. global turns off the autoprint and autoindent options and sets the report option to infinity until executing the entire command. ex sets the context mark (' ') to the current line, and changes it only if you enter open or visual mode within the global command. Default address: 1,$.
[*line1*[, *line2*]]g[lobal]! */pat/cmds* or [*line1*[, *line2*]]v */pat/cmds*	Run *cmds* globally on each line *not* matching *pat*.
[*lineno*]i[nsert]<CR>[*text*<CR>].	Insert text before *lineno*. This command differs from append only in its placement of text. Default address: current line.

Command	Description
[*lineno*]`i[nsert]!`<CR>[*text*<CR>]`.`	Same as `insert`, but changes the `autoindent` option while inserting (see ''Setting Options''). Default address: current line.
[*line1*[`,` *line2*]]`j[oin]` [*n*] [*flags*]	Join the specified lines (either the lines between *line1* and *line2* or *n* lines). `ex` ensures that there is at least one blank character where the lines joined, two if there was a period 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, it discards the white space at the start of the next line. Including a *flag* after the command (either `p` for `print`, `l` for `list`, or # for `number`) changes the display to the specified format. Default address: current line.
[*line1*[`,` *line2*]]`j[oin]!` [*n*] [*flags*]	Same as `join`, but doesn't add or delete white space. Default address: current line.
`k`*x*	A synonym for `mark` (described below), which does not require a blank or tab before the letter. Default address: current line.

Command	Description
[*line1*[, *line2*]] l[ist] [*n*] [*flags*]	Print the specified line(s), displaying tabs as ^I and the end of lines as $. Including a *flag* after the command (either p for print or # for number) changes the display to the specified format. Default address: current line.
map *string* *definition*	(Must be set via an ex command, but works in vi only.) Make typing *string* equivalent to typing *definition* when executing commands in visual mode (vi). *string* can be up to 10 characters long, and *definition* can be up to 100 characters long. Default address: none.
[*lineno*] ma[rk] *x*	Mark the specified line with *x*, a lowercase letter. You must precede *x* with a blank or a tab. After marking a line, you can refer to it with ´x. This command does not change the current line. Default address: current line.
[*line1*[, *line2*]] m[ove] *lineno*	Move the specified line(s) after *lineno*. The first moved line becomes the current line. Default address: current line.
n[ext]	Edit the next file in the argument list specified at startup. Default address: none.
n[ext]!	Move to next file and overwrite the current buffer whether you've saved it or not. Default address: none.

Command	Description
n[ext] *file-list*	Replace the current list of files to edit with the specified *file-list* and edit the first file on new list. Default address: none.
n[ext]! *file-list*	Allow editing of new *file-list* even if you have not saved the current buffer. Default address: none.
n[ext]+*cmd file-list*	Execute *cmd* (which must not have spaces in it) after opening the first file in *file-list*. Default address: none.
[*line1*[, *line2*]] nu[mber] [*n*] [*flags*]	Print the specified line(s) (*line1*, or the lines between *line1* and *line2*, or the next *n* lines) preceded by its line number. Including a *flag* after the command (either p for print or l for list) changes the display to the specified format. Default address: current line.
[*line-spec*] o[pen] [/*pat*][*flags*]	Use vi commands at each addressed line. Specifying *pat* moves the cursor to the beginning of the string matched by the pattern. Including a *flag* after the command (either p for print, l for list, or # for number) changes the display to the specified format. Type Q to exit this mode. Default address: current line.

Command	Description
pre[serve]	Save the current editor buffer as though the system had just crashed. Use this command only in emergencies when a write command results in an error and you do not know how to save your work. Use the recover command or ex -r to recover after a preserve. Default address: none.
[*line1*[,*line2*]] p[rint] [*n*]	Print the specified line(s) (*line1*, or the lines between *line1* and *line2*, or the next *n* lines) displaying nonprinting characters as ^*x* and delete (octal 177) as ^?.
[*line1*[,*line2*]] P[rint] [*n*]	Print the specified line(s) (*line1*, or the lines between *line1* and *line2*, or the next *n* lines) displaying nonprinting characters as ^*x* and delete (octal 177) as ^?.
[*lineno*] pu[t] [*buffer*]	Put previously deleted or yanked lines after *lineno*. This moves lines with delete or copies lines with yank. Specifying *buffer* (a lowercase letter between a and z) retrieves text placed in that buffer with a delete or yank command. Default address: current line.
q[uit]	Leave ex. If you haven't saved all your changes, ex warns you and doesn't allow you to leave ex. ex also tells you if there are more files in the argument list. Normally, you should write your changes before doing a quit. Default address: none.

Command	Description
`q[uit]!`	Same as `quit`, but `ex` doesn't warn you if you haven't saved the current buffer. Default address: none.
[*lineno*] `r[ead]` [*file*]	Copy the text of *file* after *lineno* in the current buffer. If you don't supply *file*, it uses the current filename. If there is no current filename, *file* becomes the current name. It will not allow you to read in devices, but it will allow you to read in binary files. If the file buffer is empty and there is no current file, `ex` treats this as an `edit` command. `0read` reads the file at the beginning of the buffer. It gives the same statistics as the `edit` command when it reads the file in. After a `read` command, the current line is the last line read (in `ex`) or the first line read (in open or visual mode). Default address: current line.
[*lineno*] `r[ead] !`*command*	Read the output of *command* into the buffer after *lineno*. There must be a blank or tab before the `!`. Default address: current line.
`rec[over]` *file*	Recover *file* after accidentally hanging up the phone, a system crash, or a `preserve` command. Default address: none.
`rew[ind]`	Start editing the files in the argument list, beginning with the first file you supplied when you started `ex`. Default address: none.

Command	Description
`rew[ind]!`	Same as `rewind`, but doesn't save the current buffer. Default address: none.
`se[t]`	The forms of this command are `set` Print those options you've changed from their default settings. `set all` Print all the option values. `set` *opt=val* Give the value *val* (either a number or a string) to the option *opt*. `set` *opt* List the current setting of a string or numeric option. `set` *opt* Turn on an option that can be either off or on. `set no`*opt* Turn off an option that can be either off or on. `set` *opt*? List the current setting of an option that can be either off or on. You can give more than one parameter to `set`; parameters are interpreted left-to-right. See ''Option Descriptions'' for the complete list. Default address: none.
`sh[ell]`	Create a new shell. Editing resumes when you terminate the new shell (using `exit`). Default address: none.
`so[urce]` *file*	Read *file* and execute the (text-manipulation) commands in it. You can nest this command. Default address: none.

Command	Description
[*line1*[`,` *line2*]] `s[ubstitute]`/*pat*/*repl*[`/`*suffix*]	
	`s` replaces the first instance of pattern *pat* with replacement pattern *repl* on each specified line. The *suffixes* are
	`g` (global.) Substitute *pat* with *repl* every time it appears on the specified lines. To make the substitution everywhere in the file, use the format `1,$s/`*pat*`/`*repl*`/g`. You can also use `%` instead of `1,$`.
	`c` (confirm.) Print the line before making each substitution, marking the string to substitute with `^` characters. Type `y` to confirm the substitution, and type any other character if you don't want to make the substitution.
	`r` (replace.) Replace the previous replacement pattern from a substitution with the most recently mentioned regular expression; for example, from a search command.
	You can split lines by substituting newline characters into them. You must escape the newline in *repl* by preceding it with a backslash (\). See ''Regular Expressions and Searching'' for other metacharacters available in *pat* and *repl*. Default address: current line.
[*line1*[`,` *line2*]] `s[ubstitute]` *suffix*	
	Omitting *pat* and *repl* repeats the last substitution. This is a synonym for the `&` command, which is described later in this section.

Command	Description
	Using the `r` suffix (`sr`) replaces the previous *pat* with the previous regular expression. This is a synonym for the ~ command, which is described later in this section. Default address: current line.
[*line1*[, *line2*]] `t` *lineno flags*	`t` is a synonym for `copy`.
`ta[g]` *tag*	Start editing the file at *tag*, moving to another file, if necessary. You must write the current file, if you modified it, before giving a `tag` command. If you give the `tag` command without specifying a *tag*, it uses the previous tag. Normally you use this command after using the `ctags`(1) command to create a tag file. (See `ctags`(1) in *A/UX Command Reference*.) Default address: none.
`una[bbreviate]` *wd*	Delete *wd* from the list of abbreviations. When you type *wd*, it is not expanded.
`u[ndo]`	`undo` reverses the changes made by the last editing command, except `write` or `edit`. `undo` marks the previous current line with ´´. If you restored a line, this becomes the current line. If you didn't restore a line, the line before the last deleted line becomes the current line.

Command	Description
`unm[ap]` *string*	Reverse the effect of a previous `map` command, removing the definition associated with *string*. (Note that the `map` command only affects visual mode.) Default address: none.
[*line1*[, *line2*]]`v` /*pat*/*cmds*	A synonym for the variant form of a `global` command: runs *cmds* at each line *not* matching *pat*. Default address: none.
`ver[sion]`	Give the current version of the editor and the last date the editor was changed. Default address: none.
[*line-spec*] `vi`[*type*][*n*][*flags*]	Enter visual mode at the specified line. The optional *type* argument (- ^ or .) specifies where the line is placed on the screen. If you omit *type*, the specified line is the first line on the screen. *n* specifies an initial window size; default is the value of the option `window`. Type `Q` to exit this mode. Default address: current line.
[*line1*[, *line2*]] `w[rite]` [*file*]	`write` writes changes back to *file*, printing the number of lines and characters written. Normally, you omit *file* and the text goes back where it came from. If you specify *file*, text is written to that file. By default, it writes the entire file.

Command	Description
	The editor writes to a file only if it is the current file, if it is creating the file, or if the file is actually a device (`/dev/tty`, `/dev/null`). Otherwise, you must give the variant form `w!` to force the write. If the file does not exist, `ex` creates it. This command does not change the current line. If there is an error while writing the current and edited file, the editor considers that there has been no write since the last change, even if the buffer had not previously been modified. Default address: current line.
[*line1*[`,` *line2*]] `w[rite]` >> *file*	Append buffer contents to *file*. Default address: current line.
`w[rite]!` *file*	Force a write to a file. This is helpful when you want to write to a file that already exists. Default address: none.
[*line1*[`,` *line2*]] `w[rite]` `!`*command*	Write the specified line(s) into *command*. Note that this is different from `w!` because a blank or tab must separate the `w` from the `!`. Default address: current line.
`wq` [*file*]	`write` followed by `quit`. Default address: none
`wq!` [*file*]	The variant overrides checking on the `write` command, as `w!` does.

Command	Description
`x[it]` *[file]*	Write the buffer if there have been any changes, then quit the file. Default address: none.
[line1`[,`*line2]]* `ya[nk]` *buffer n*	`yank` places a copy of the specified line(s) in the named buffer. You can retrieve them with `put`. If you don't specify a buffer name, the lines go to a more volatile place (see the `put` command description). Default address: current address.
[lineno+1] `z` *n*	`z` prints the next *n* lines (default `window`).
[lineno] `z`*type n*	The `z` command determines where the current line appears on the screen. *type* is the character following the command and determines the positioning of the display on the screen. There are several different forms of this command: *[lineno]*`z` or *[n]*`z+` Print the next screenful of lines with the current (or specified) line at the top of the screen. *[lineno]*`z-` Print the screen with the current (or specified) line at the bottom. *[lineno]*`z.` Print the screen with the current (or specified) line at the center. *[lineno]*`z=` Print the screen with the current (or specified) line in the center, surrounded by lines of – characters.

Command	Description
	[*lineno*]z^ Print the screen two windows before the current (or specified) line. Default address: current line.
! *command*	Send *command* to the shell to execute. Within *command,* % and # are expanded as in filenames; ! is replaced with the text of the previous command. Thus, !! repeats the last shell escape. If there is any such expansion, the expanded line is echoed. This command does not change the current line. If you haven't written the buffer contents since the last change, ex prints a warning message before executing the command. A single ! prints when the command completes. Default address: current line.
[*line1*[, *line2*]] ! *command*	Supply the specified address (or address range) as standard input to *command.* The output then replaces the input line(s). Default address: current line.
[*line-spec*]=	Print the line number of the specified line. ''Dot equals'' (.=) gives the current line number. If no line is specified, the line number of the last line in the file is given. Does not change the current line. Default address: last line in file.

Command	Description
[*line1*[, *line2*]] < *n flags* or [*line1*[, *line2*]] > *n flags*	The less-than and greater-than signs (< and >) shift left or right a distance specified by the `shiftwidth` option. They shift only blanks and tabs and do not discard nonwhite characters in a left shift. The current line is the last line that changed due to the shifting. Default address: current line.
[*line-spec*]CONTROL-d	CONTROL-d scrolls through the file. You can specify the size of the scroll with the `scroll` option. The default is a half screen of text. Default address: current line.
[*line1*[, *line2*]] or <CR>	Print the addressed line(s). Pressing RETURN prints the next line. Default address: none.
[*line-spec*] &*suffix n flags*	The ampersand (&) repeats the previous substitute command on the current (or specified) line. If you set the `edcompatible` option, it retains the suffix; that is, if the previous substitute command was global, the ampersand repeats the substitution globally on the current line. Default address: current line
[~ [*line-spec*]] ~*suffix n flags*	The tilde (~) replaces the previous replacement pattern from a substitution with the previous regular expression. Default address: current line

Command	Description
[*line1*[, *line2*]] # [*n*] [*flags*]	Print the specified line(s) (*line1*, or the lines between *line1* and *line2*, or the next *n* lines) preceded by its line number. Including a *flag* after the command (either `p` for `print` or `l` for `list`) changes the display to the specified format. Default address: current line.

Chapter 6
Using sed

Contents

Chapter 6

Using sed

1. Introduction

The sed editor is a stream editor. It is especially useful for

- Editing large files that cannot be contained in a buffer. The size of a file to be edited with sed is limited only by the amount of secondary storage. Only a few lines of the current input file are in physical (volatile) memory at one time, and no temporary files (buffers) are used.
- Performing complicated editing sequences on any size file. The sed editor is most commonly used in shell scripts, where complicated editing requests can be stored, edited, and applied to the input file(s) as a command.
- Efficiently performing multiple global editing commands in one pass. The sed program running from a command file is faster and more efficient than an interactive editor like ex, even when ex is also running from a command file.
- Performing transformations on a data stream as part of a pipe or a shell script.

Note that sed does not recognize certain commands provided by an interactive editor. For example, sed does not provide relative addressing. Because it operates on one line at a time, sed cannot move backward or forward relative to the current line in a file. In addition, sed does not inform you about the effects of your commands, or allow you to undo them immediately.

2. Overall operation

By default, sed copies standard input to standard output, performing zero or more editing actions on each line before writing it to the output. Editing actions are specified by sed editing commands, described in the next section. You specify the lines to be affected by these commands by addresses, either context addresses or line numbers.

You never modify an input file directly; instead, changes are written to the standard output. If this output is redirected to a file, then the new file contains the modifications created by your editing actions. Then you may, if you wish, replace the original file with this new file.

2.1 Command line options

Command syntax for the `sed` editor is

`sed [-n] -e` *'command-line-script'* `[-e` *'command-line-script'* `]...`
  `[-f` *sfile*`]...[`*file*`...]`

or

`sed [-n] -f` *sfile*`...[`*-f sfile*`]...[`*-e 'command-line-script'*`]...[`*file*`...]`

`sed [-n]` *'command-line-script'* `[`*file*`...]`

`sed` must be invoked with at least one `-e` or `-f` option; however, if only ``-e *'command-line-script'* '' is used, the `-e` may be omitted.

`sed` can be invoked in one of the following ways:

```
sed 'command-line-script' file
sed -e 'command-line-script' file
sed -n -e 'command-line-script' file
sed -f sfile file
sed -n -f sfile file
sed -e 'command-line-script' -f sfile file
```

The options are as follows:

`-n` (no-copy.) Copy only those lines explicitly specified either by `p` (print), `i` (insert), or `a` (append) commands or `p` arguments after `s` (substitute) commands.

`-e` (expression.) The *command-line-script* argument is an ``expression'' (inline `sed` command(s) using the syntax of regular expressions and enclosed in single or double quotes) to be run on the input stream. There may be more than one `-e` (with its corresponding expression) on a command line. If the newlines are escaped, there may be more than one line in an expression. The `-e` itself may be omitted if there is only one expression and no `-f` option is present.

`-f` (source file.) The *sfile* argument is a file containing `sed` commands, one to a line. There may be more than one `-f` option specified on the command line. If multiple `-f` command file arguments are given, the commands they contain are executed in the order specified.

The input *files* may be omitted; in that case, `sed` takes its input from the standard input. Note that `sed` does not accept the "–" construct used in other programs (for example, `awk`) to indicate the standard input. If you must apply a sequence of `sed` commands to some files and then to the standard input, you can use the following command:

```
cat files - | sed -f sfile
```

2.2 Usage

Editing commands are specified on the `sed` command line. They can either be embedded inline (with the `-e` option) or enclosed in a file and provided as an argument to the `-f` option. The following are examples of `sed` usage:

```
sort chap.1 |
sed -e 's/\.dc\./.dec./' -e 's/\.3b\./.u3b./'
```

This sorts the contents of `chap.1` and performs substitutions on the first instance of `.dc.` and `.3b.` in each line; the results are written to standard output.

Note that

```
sed -e 's/\.dc\./.dec./' -e 's/\.3b\./.u3b./'
```

is equivalent to

```
sed '
    s/\.dc\./.dec./
    s/\.3b\./.u3b./
'
```

In this chapter, we use the first form, which employs the `-e` option. These may be replaced with the second form if you prefer.

With `-e`, you may also separate editing commands with a semicolon. For example,

```
sed -e 's/\.dc\./.dec./;s/\.3b\./.u3b./'
```

is equivalent to the above examples.

The command form:

```
sed -e 's/\.dc\./.dec./
s/\.3b\./.u3b./' chap.1
```

performs the same substitutions as the preceding command on a file named `chap.1`; the results are written to standard output.

> *Note:* When using `sed` in the C shell, newlines must be escaped using a backslash even when enclosed in single quotes.

The command:

```
sed -e 's/,/ /g' chap.1 > temp
```

replaces every comma (`,`) in `chap.1` with a space; the modified file is contained in `temp`.

If you put the following `sed` commands into a file named `cmd.file`:

```
s/\.dc/.dec./g
s/\.3b\./.u3b./
s/,/ /g
```

then you can use the following command:

```
sed -f cmd.file chap.1 > temp
```

This performs substitutions on `chap.1`; the modified file is contained in `temp`.

You can also use the command

```
sed -n -f cmd.file chap.1
```

to perform substitutions on `chap.1` and write the modified `chap.1` to standard output.

Before any input file is opened, all editing commands are compiled in the order encountered (also the order in which they are attempted at execution time) into a form that will be moderately efficient during the execution phase. In the execution phase the commands are actually applied to lines of the input file.

2.3 Editing command syntax

The general editing command syntax is

[line-spec] command [arguments]

The *line-spec* (line specification) and the *arguments* are optional, although either of these may be required according to the command given. *line-specs* may be line number(s) or context addresses in the form

[*line1* [, *line2*]]

or

[/*pattern*[/][, /*pattern*[/]]

In the first case, if one line number is specified, `sed` performs the editing command on that line; if two line numbers are specified, `sed` performs the editing command on the range of lines between *line1* and *line2*, inclusive. In the second case, if one context address is specified, `sed` performs the editing command only on lines containing that *pattern*; if two context addresses are specified, `sed` performs the editing command on all lines between the first *pattern* and the second *pattern*, inclusive. After it recognizes the second pattern, `sed` searches for the first pattern again. If found, it begins performing the editing command again until it recognizes the second pattern, and so on. Any number of blanks or tabs may separate *line-specs* from the command; blanks and tab characters at the beginning of lines are ignored.

2.4 Command application order

Commands are applied one at a time, in the order encountered; the input to each command is the output of all previous commands.

This default linear ordering can be changed by the `t` (test substitution) and `b` (branch) control-flow commands. When the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied commands.

2.5 Pattern space

The **pattern space** is the buffer the `sed` commands operate on. Ordinarily, pattern space is one line of the input text, but more than one line can be read into the pattern space by using the `N` command or the `G` command.

3. Addressing

Input file lines to be affected by your editing commands are specified by *line-specs*. These *line-specs* can be either line numbers or context addresses. If no *line-spec* is present, the command is applied to every line in the input file.

Multiple commands can be applied to a single *line-spec* by grouping commands with braces in the following format:

```
line-spec  {
                command-list
           }
```

3.1 Line number addresses

A line number is a positive decimal integer that is incremented (by an internal counter) as each line is read from the input. A **line number address** corresponds to the value of the internal line counter. As a special case, the `$` character matches the last line of the last input file.

> *Note:* The line counter runs cumulatively through multiple input files. It is not reset when a new consecutive input file is opened.

Commands can be preceded by zero, one, or two addresses. It is an error when a command has more addresses than allowed.

If a command has zero 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 separated by a comma, it is applied to the first line that matches the first address and to all subsequent lines up to, and including, the first line that matches the second address. An attempt is made on subsequent lines to match the first address again, and the process is repeated.

3.2 Context addresses

A context address is a regular expression enclosed by matching delimiters. Any character may be selected as the expression delimiter (for example, */pattern/* or `%`*pattern*`%`). `sed` recognizes regular expressions that have the following construction:

- An ordinary character is a regular expression and matches that character.
- A caret (`^`) at the beginning of a regular expression matches the beginning of a line.
- A dollar sign (`$`) at the end of a regular expression matches the end of a line.
- The (`\n`) character matches an embedded newline character in the pattern space but not the newline character at the end of the pattern space. Newlines may be embedded by using the `N` command or the `G` command.
- A period (`.`) matches any character except the terminal newline character of the pattern space.
- A regular expression followed by an asterisk (`*`) matches any number (including zero) of adjacent occurrences of the regular expression it follows.
- A string of characters in square brackets (`[]`) matches any character in the string and no others. For example, `[abc]` matches the single-character strings `a`, `b`, and `c`. The characters may also be specified as a range using the format

  ```
  [a-z]
  ```

  which will match any lowercase character. If, however, the first character of the string is a caret (`^`), the regular expression matches any character except the characters in the string and the terminal newline character of the pattern space. The caret is the only metacharacter recognized within the square brackets. If (`]`) needs to be in the string enclosed in square brackets, it should be the first non-metacharacter. Thus, for example,

`[]...]`	includes `]`
`[^]...]`	does not include `]`

  In both cases, a range may be specified by using a hyphen (for example, `[A-Z]` or `[0-9]`).
- A concatenation of regular expressions is a regular expression that 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 under the substitute command (`s`) later in this section.

- The expression \d (where *d* is a digit, 0 through 9) refers to the string of characters found earlier in the same pattern by an expression using the `\(` and `\)` construction. The `\(` and `\)` sequences act just like those in the other A/UX editors, and are used to establish ''fields'' or sections in a line of text (or all lines of text) in a file.

  For example, suppose a file contained the following list of names:

  ```
  Dick Powell
  William Powell
  Eleanor Powell
  Jane Powell
  ```

  The following expression reverses the order of the names, while placing a comma and a space between each first name and last name:

  ```
  s/\([A-Z].* \)\([A-Z].*\)/\2, \1/
  ```

  This command writes a new list:

  ```
  Powell, Dick
  Powell, William
  Powell, Eleanor
  Powell, Jane
  ```

  For another example, the following expression matches a line beginning with two repeated occurrences of the same string separated by a space:

  ```
  /^\(.*\) \1/
  ```

- A null regular expression standing alone (for example, `//`) is equivalent to the previous regular expression.

- Special characters `^`, `$`, `*`, `\`, and `/`, when used as literal characters, must be preceded by a backslash (`\`).
- For a context address to match, the whole pattern within the input address must match some portion of the pattern space.

3.3 Examples

Let us consider more examples of using `sed`. First, create a text file named `poem` that contains the following 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.
```

Examples in this chapter use this text except where noted. The following shows the output of a `sed` command using line number addressing. The command

```
sed -e '2q' poem
```

copies the first two lines of the input and quits. The output on your screen will be

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

On the same input text, the following lists the matches resulting from several `sed` commands using context addressing:

`/an/`	matches lines 1, 3, and 4
`/an.*an/`	matches line 1
`/^an/`	matches no lines
`/./`	matches all lines
`/\./`	matches line 5
`/r*an/`	matches lines 3 and 4
`/\(an\).*\1/`	matches line 1

4. sed editing commands

In the following summary, *line-spec* indicates a single line number or a context address. *line1*, *line2* indicates a range of addresses from *line1* to *line2*. If you don't specify an address, the commands are applied to all lines in the file (unless otherwise noted).

4.1 Line-oriented commands

The commands in this section apply to the entire line (or lines) currently stored in the pattern space.

Command	Description
[*line1*[, *line2*]]d	(delete.) The d command deletes from the file (does not write to the output) those lines matched by its addresses. It also has the side effect that no further commands are attempted on the corpse of a deleted line. As soon as the d command is executed, a new line is read from the input, and the list of editing commands is restarted from the beginning on the new line.
[*line1*[, *line2*]]n	(next.) The n command reads the next line from the input, replacing the current line and incrementing the internal line counter. The current line is written to standard output. The list of editing commands continues following the n command.
[*line-spec*]a\<CR>*text*	(append.) The a command causes the *text* argument to be written to the output after the line matched by its address. The a command is inherently multiline; a must appear at the end of a line, and *text* may contain any number of lines. To preserve the one-command-to-a-line fiction, interior newline characters must be hidden by a backslash character (\) immediately preceding the newline character.

Command	Description
	The *text* is terminated by the first newline character not immediately preceded by a backslash. Once an `a` command is successfully executed, text will be written to the output regardless of what later commands do to the line that triggered it. Even if that line is deleted, text will still be written to the output. The *text* is not scanned for address matches, and no editing commands are attempted on it. The `a` command does not cause a change in the line number counter.
[*line-spec*]`i\`<CR>*text*	(insert.) The `i` command causes the *text* argument to be written to the output before the line matched by its address. The `i` command is inherently multiline; `i` must appear at the end of a line, and *text* may contain any number of lines. To preserve the one-command-to-a-line fiction, interior newline characters must be hidden by a backslash character (\) immediately preceding the newline character. The *text* is terminated by the first newline character not immediately preceded by a backslash. Once an `i` command is successfully executed, text will be written to the output regardless of what later commands do to the line that triggered it. Even if that line is deleted, text will still be written to the output. The *text* is not scanned for address matches, and no editing commands are attempted on it. The `i` command does not cause a change in the line number counter.

Command	Description
[*line1*[, *line2*]]`c\`<CR>*text*	(change.) The `c` command deletes lines selected by its addresses and replaces them with the lines in the *text* argument. Like `a` and `i`, `c` must be followed by a newline character hidden by a backslash; interior newline characters in *text* must be hidden by backslashes. The `c` command may have two addresses and therefore select a range of lines. If it does, all lines in the range are deleted, but only one copy of *text* is written to the output, not one copy per line deleted. As with `a` and `i`, *text* is not scanned for address matches, and no editing commands are attempted on it. It does not change the line number counter. After a line has been deleted by a `c` command, no further commands are attempted on the corpse. If text is appended after a line by `a` or `r` commands and the line is subsequently changed, the text inserted by the `c` command will be placed before the text of the `a` or `r` commands. (The `r` command is described later.)

Leading blanks and tabs disappear from text inserted in the output by the `a`, `i`, and `c` commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash. The backslash will not appear in the output.

The following example shows line-oriented `sed` commands used on the standard input example shown in Section 3.3.

If the file `script` contains the lines

```
n
a\
XXXX
d
```

the command

```
sed -f script poem > outputfile
```

produces an *outputfile* that contains the following lines:

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

4.2 The substitute command

The substitute command uses the following syntax:

[*line1*[, *line2*]]s *pattern replacement flags*

The `s` command replaces the part of a line selected by *pattern* with *replacement*. It can be read "substitute for *pattern, replacement*." The command arguments are described as follows:

pattern
: The *pattern* argument is a regular expression, like the patterns in context addresses. The only difference between *pattern* and a context address is that the context address must be delimited by slash (/) characters; *pattern* may be delimited by any character other than space or newline. By default, only the first string matched by *pattern* is replaced unless the `g` flag (below) is invoked.

replacement
: 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 that are special in patterns do not have special meaning in *replacement*. Instead, the following other characters are special:

& is replaced by the string matched by *pattern*.

\d is replaced by substring *d* (*d* is a single digit), matched by parts of *pattern*, and enclosed in `\(` and `\)`. If nested substrings occur in *pattern*, substring *d* is determined by counting opening delimiters (`\(`). As in patterns, special characters may be made literal characters by preceding them with a backslash (\\).

flags

The *flags* argument may contain the following:

`g` (global.) Substitute *replacement* for all nonoverlapping 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.) Print the line if a successful replacement was done. The `p` flag causes the line to be written to the output if a substitution was actually made by the `s` command. If several `s` commands, 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. Note that unless the `-n` flag option is used, each line will be echoed automatically to standard output. In addition, each line affected by the `p` flag will be echoed as well, causing multiple copies to be written to standard output.

`w` *file*

(write to file.) Write the line to a file if a successful replacement was done. A single space must separate `w` and *file*. The `w` flag causes lines that are actually substituted by the `s` command to be written to a file named by *file*. If *file* exists before `sed` is run, it is overwritten; if not, it is created. The possibilities of multiple, somewhat different copies of one input line being written are the same as for p. A maximum of ten different filenames may be mentioned after w flags and w commands.

The command

```
cat poem | sed -e 's/to/by/w changes'
```

produces an output file named changes that contains only these lines that were changed:

```
Through caverns measureless by man
Down by a sunless sea.
```

If the no-copy option is in effect (using the -n option on the sed command line), then the same effect can be achieved with the command

```
sed -n -e 's/to/by/p' poem > changes
```

If your command file script contains the line

```
s/[\.,;?:]/*P&*/gp
```

then the command

```
sed -n -f script poem
```

produces the output

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

If the g flag is not used, the substitution takes effect only on the first instance of the pattern in a given line. For example, the command

```
sed -n -e '/X/s/an/AN/p' poem
```

causes the substitution to occur only on the first instance of an

```
In XANadu did Kubla Khan
```

4.3 Input/output commands

Command	Description
[*line1* [, *line2*]]p	(print.) The p command writes addressed lines to the standard output file. They are written at the time the p command is encountered, regardless of what succeeding editing commands may do to the lines.
[*line1* [, *line2*]]w *file*	(write to file.) The w command writes addressed lines to the file named by *file*. Exactly one space must separate the w and *file*. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write command is encountered for each line, regardless of what subsequent editing commands may do to them. A maximum of ten different files may be mentioned in write commands and w flags after s commands combined.
[*line-spec*]r *file*	(read from file.) The r command reads the contents of *file* and appends them after the line matched by the address. Exactly one space must separate the r and *file*. The file is read and appended regardless of what subsequent editing commands may do to the line that matched its address. If r and a commands are executed on the same line, the text from a commands and r commands is written to the output in the order that the commands are executed. If a file mentioned by an r command cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

Note: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in `w` commands or flags. That number is reduced by one if any `r` commands are present (only one read file may be opened at a time).

If 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 command

```
sed -e '/Kubla/r note1' poem
```

produces

```
In Xanadu did Kubla Khan
Note: Kubla Khan (more properly Kublai Khan;
1216-1294) was the grandson and most eminent
successor of Genghiz (Chingiz) Khan and founder
of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

4.4 Multiple input line commands

The following three commands, all spelled with uppercase letters, deal with pattern spaces containing embedded newline characters. They are intended principally to provide pattern matches across lines in the input. The `P` and `D` commands are equivalent to their lowercase counterparts if there are no embedded newline characters in the pattern space.

Command	Description
[*line1*[, *line2*]]N	Append the next input line to the current line in the pattern space. The two input lines are separated by an embedded newline character. Pattern matches may extend across embedded newline characters.
[*line1*[, *line2*]]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 character was the terminal newline character), read another line from the input. In any case, begin the list of editing commands again from the beginning.
[*line1*[, *line2*]]P	Print the first part of the pattern space. Print up to, and including, the first newline character in the pattern space.

4.5 Hold and get commands

The following commands save and retrieve part of the input for possible later use.

Command	Description
[*line1*[, *line2*]]h	Hold pattern space. The h command copies the contents of the pattern space into a hold area, destroying the previous contents of the hold area.
[*line1*[, *line2*]]H	Hold pattern space. The H command appends contents of the pattern space to contents of the hold area. Former and new contents are separated by a newline character.

Command	Description
[*line1*[`,` *line2*]]`g`	Get contents of hold area. The `g` command copies contents of the hold area into the pattern space, destroying previous contents.
[*line1*[`,` *line2*]]`G`	Get contents of hold area. The `G` command appends contents of the hold area to contents of the pattern space. Former and new contents are separated by a newline character.
[*line1*[`,` *line2*]]`x`	Exchange. The `x` command interchanges contents of the pattern space and the hold area.

For example, if your `sed` command file contains the commands

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

when applied to the file `poem`, this produces

```
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
```

4.6 Control-flow commands

These commands do no editing on the input lines but control the application of commands to the lines selected by the address part.

Command	Description
[*line1*[, *line2*]] !	(don't.) The ! command causes the next command (written on the same line) to be applied to those input lines not selected by the address part.
[*line1*[, *line2*]] {	(grouping.) The { command causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the { or on the next line. The group of commands is terminated by a matching } standing on a line by itself. Groups can be nested.
: *label*	(place label.) The : command marks a place in the list of editing commands that may be referred to by `b` and `t` commands. The *label* argument may be any sequence of eight or fewer characters. If two different colon commands have identical labels, a compile time diagnostic will be generated and no execution attempted.
[*line1*[, *line2*]]`b` *label*	(branch to label.) The `b` command causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon command with the same *label* was encountered. The space between the `b` command and the *label* is optional. If no colon command with the same label can be found after all editing commands have been compiled, a compile time diagnostic is produced and no execution is attempted.

Command	Description
	A `b` command with no *label* is 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.
[*line1*[`,` *line2*]]`t`*label*	(test substitutions.) The `t` command tests whether any successful substitutions have been made on the current input line; if so, it branches to *label;* if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by reading a new input line or by executing a `t` command.

4.7 Miscellaneous commands

Command	Description
[*line-spec*]=	The = command writes the line number of the line matched by its address to the standard output.
[*line-spec*]`q`	The `q` command causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

1027046