Burroughs

XE 500 CENTIX™

Operations Guide

Volume 2: Editing Operations

Relative To Release Level 6.0
Priced Item
December 1986

Distribution Code SA
Printed in U S America
1207909

Burroughs

XE 500 CENTIX™

Operations Guide

Volume 2: Editing Operations

Relative To Release Level 6.0
Priced Item
December 1986

Distribution Code SA
Printed in U S America
1207909

About This Guide

Purpose

This guide describes how to operate the various editors that are available on the CENTIX operating system. Since much of the information is presented in tutorial sessions, each reader is encouraged to participate with them.

Scope

This guide describes the various editors that are provided by the CENTIX operating system. If you are unfamiliar with the CENTIX operating system, refer to the *XE 500 CENTIX Operations Guide, Volume 1: Basic Operations*. For complete documentation on CENTIX operating system capabilities, refer to the *XE 500 CENTIX Operations Reference Manual.*

Audience

The audience for this guide is the novice CENTIX user who wants to learn to use any of the various editors which are available on the CENTIX operating system. Sophisticated CENTIX users may find useful information concerning some of the more advanced capabilities of the CENTIX editors.

Prerequisites

The user who reads this guide should at least be familiar with the information contained in the *XE 500 CENTIX Operations Guide, Volume 1: Basic Operations.*

How to Use This Document

Use this guide to learn how to edit files with the CENTIX editors. Again, all readers are encouraged to participate in the many tutorial sessions found throughout this guide.

Organization

This guide contains the following sections:

Section 8, Introduction to CENTIX Editing Operations, describes the various editors available on the CENTIX operating system.

Section 9, Text Editor (ed)—Basic, provides an introduction to how to use the text editor (ed) for creating and changing text.

Section 10, Text Editor (ed)—Advanced, describes using some of the more complex capabilities of the text editor (ed).

Section 11, Visual Editor (vi)—Basic, provides an introduction to the visual editor (vi).

Section 12, Visual Editor (vi)—Advanced, describes using some of the more complex capabilities of the visual editor (vi).

Section 13, Text Editor (ex), describes using the text editor (ex).

Section 14, Stream Editor (sed), describes using the stream editor (sed) and discusses instances in which it is preferable to using the text editor (ed).

Section 15, Big File Scanner (bfs), describes using the big file scanner (bfs).

Section 16, Editing BTOS files with vi and ed, describes how to access and edit BTOS files from a PT 1500 terminal using either the vi or ed editor.

An index follows Section 16.

Related Product Information

XE 500 CENTIX Administration Guide

This guide discusses how to administer the XE 500 CENTIX operating system.

XE 500 CENTIX centrEASE Operations Reference Manual

This manual describes how to use the CENTIX administrative facility, centrEASE.

XE 500 CENTIX C Language Programming Reference Manual

This manual describes the programming language on which the CENTIX operating system is structured, C.

XE 500 CENTIX Operations Guide, Volume 1: Basic Operations

This manual describes the basic operations of the CENTIX system and provides an overview for the novice CENTIX system user.

XE 500 CENTIX Programming Guide

The guide discusses how to program on the XE 500 CENTIX system.

XE 500 CENTIX Operations Reference Manual

This manual lists and describes all CENTIX shell commands, system calls, library functions, and special files.

Conventions Used in This Guide

- All commands within text are shown in boldface.
- Variables are shown in italics. For example, in the following command, *filename* is a variable:

  ```
  $ vi filename
  ```

  When you enter the command, you substitute the actual name of the file for *filename*.
- In command lines, optional fields are enclosed in brackets.

Contents

Introduction to CENTIX Editing Operations

The CENTIX operating system provides several different methods for writing and storing text or computer programs into files. The following editor programs are provided by the CENTIX operating system and are presented in tutorial fashion in this guide.

- Text editor (**ed**) is an interactive line oriented editor with commands displayed interspersed with text.
- Visual editor (**vi**) is a full screen, interactive editor with **ex** editor commands available at the bottom of the screen (**vi** is a superset of **ex**).
- Text editor (**ex**) is an interactive line oriented editor with commands displayed at the bottom of the screen.
- Stream editor (**sed**) is a noninteractive editor using commands that operate on one line at a time.
- Big file scanner (**bfs**) is used to scan (read-only) very large files.

Text Editor (ed)--Basic

The **ed** program is a text editor. A text editor is an interactive program for creating and changing text, using directions (commands) provided by you at a terminal. The text is often a document like this one or perhaps data for a program.

Text can be input with the standard text editor **ed**. This section covers enough for your day-to-day needs.

The **ed** program does all editing actions in a temporary file called a buffer. The contents of a file to be edited are first read into the buffer. Changes made in the buffer have no effect on the original file until the buffer contents are written to the original file via the **w** (**write**) command. If you leave the **ed** program without writing, all text in the buffer is lost. After the buffer contents are written, the previous contents of the original file are no longer accessible.

The **ed** program operates in two modes, as follows:

- Command mode
- Input mode

When you first enter the **ed** program, you are placed in the command mode by default. While in the command mode, the **ed** program accepts editing commands. Editing commands are single letter commands that have the following format:

[*address*] *command*

The bracket characters ([]) above indicate that *address* is an optional field that specifies a line range in the buffer on which the *command* will execute. A single number address refers to one particular line in your buffer. Two numbers separated by a comma refer to a range of lines in your buffer. If no address is given, the current line is the default. Since every command that requires addresses has default addresses, the addresses can often be omitted. The editing commands are summarized in part "Summary of Commands and Line Numbers" found in this section.

The **ed** program is placed in the input mode by invoking (executing) certain editing commands such as **append (a)**, **change (c)**, and **insert (i)**. While **ed** is accepting text, it is in the input mode. No commands are recognized when you are in the input mode. The input mode is left by entering a period (.) alone at the beginning of a line and pressing the RETURN key.

Getting Started

Assume that you have logged in to a CENTIX system and it has just printed the prompt,

```
$
```

Enter the following command to create a new file or edit an existing file (remember you must press the RETURN key to enter each command):

```
$ ed filename
```

If the *filename* already exists, **ed** will print the number of characters it wrote to the buffer. If you are creating a new file, the system will then prompt you with

```
?filename
```

This output means you have specified a file that does not exist in your current directory. You can now type in text using the **a (append)** command.

Creating Text (a)

As your first problem, suppose some text is to be created starting from scratch. Perhaps the first draft of a document or paper is to be entered. Normally, it will have to start somewhere and undergo changes (editing) later. This subsection describes how to enter some text to get a file of text started. How to make changes and corrections to the text is described later.

When **ed** is first invoked, it is like working with a blank sheet of paper (the file)-there is no text or information present on the paper (in the file). The text must be supplied by the person using **ed**; it is usually done by typing in the text or by reading it into **ed** from a file. We will start by typing in some text and return shortly to how to read files.

First a bit of terminology. In **ed** jargon, the text being worked on is said to be "kept in a buffer." Think of the buffer as a blank page, if desired, or simply as the information that is to be edited. In effect the buffer is like the sheet of paper where you will write things, then change some of them, and finally file the whole thing away.

You tell **ed** what to do to the text by typing instructions called commands. Most commands are single lowercase letters, sometimes preceded by information about the specific lines affected. Each command is typed on a separate line and ended by pressing the RETURN key. The **ed** text editor makes no response to most commands-there is no prompting or response messages like "ready." This silence is preferred by some users. Once in the editor, beginners should enter

```
P
```

to get an input prompt and enter

```
H
```

for help.

Note: *When starting a new file and immediately entering **H**, the system will return the following message "cannot open input file." Ignore this message for this situation. The message will not be received when you ask for help after entering an existing file.*

The prompt or help can be turned off by entering the **P** or **H** a second time.

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

```
a
```

on a command line; then press the RETURN key. It means "append (or add) text lines to the buffer as you type them in." Appending is like writing fresh material on a piece of paper. So enter lines of text into the buffer as follows (remember to press the RETURN key after finishing each line):

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

The way to stop appending is to type a line that contains only a period (.). The "." is used to tell **ed** that the appending is finished. (Even experienced users forget to end appending with a "." sometimes. If **ed** seems to be ignoring your entries, type an extra line with just the "." on it. You may then find you have added some unnecessary lines to your text that you will have to take out later.)

After the **append** command has been used, the buffer will contain the following three lines:

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

The **a** and the "." are not there because they are not text.

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

Obtaining Error Messages (?)

If an error occurs in typing commands into **ed**, the text editor will tell you by typing the following:

```
?
```

Experienced users can figure out most problems at a glance. You can get a brief explanation of the error by typing

```
h
```

Remember to complete all commands by pressing the RETURN key. The help command gives a short error message that explains the reason for the most recent **?** diagnostic. If you want the error message printed without having to enter an **h** every time, enter

```
H
```

Writing Text (w)

You may want to save your text for later use. To write out the contents of the buffer onto a file, use the **write** command followed by the file name to write on.

```
w filename
```

This will copy the buffer's contents onto the specified *filename* (destroying any previous information in the file). For example, to save (write) the text in a file named *test*, enter

```
w test
```

Leave a space between **w** and the file name.

If you specified a file on entering the editor (**ed** test), that file (test) is written to by default by entering

```
w
```

The **ed** program will respond for both cases by printing the number of characters it wrote out. The **ed** program would respond with

```
71
```

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

The **ed** program works on a copy of the original file that is stored in a buffer, not the original file itself. No change in the contents of the original file takes place until you give a **w** **(write)** command. Writing out the text onto a file from time to time as it is being created is a good idea.

Exiting ed (q)

To end a session with **ed**, first save your text by writing it onto a file using the **w** **(write)** command, and then type the **q** **(quit)** command

```
q
```

The system will respond with the prompt character

```
$
```

At this point, your buffer vanishes with all its text. This is why you would want to write before quitting. Actually, **ed** will print the character

```
?
```

if you try to quit without writing. At this point, you write if desired; if not, another **q** will get you out regardless and will not save the text in the buffer.

Example One

Enter **ed**, create a file called *test*, and input some text using the **append** command (**a**) as follows:

```
$ ed test
?test
a
Now is the time
for all good people
to come to the aid of their party.
.
w
71
q
```

Note that no system prompt appears while in the text editor. Do not forget to write the text into a file with the **write** command (**w**). You left **ed** by using the **q** command. If you are in the shell environment, print the file to see that everything worked. To print a file on the screen, enter

```
$ cat test
```

in response to the prompt character ($), and the file test is displayed as is shown below:

```
$ cat test
Now is the time
for all good people
to come to the aid of their party.
```

Reading Text (r)

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

```
r test
```

will read the file test into the buffer. The command appends the file specified to the end of whatever file is already in the buffer. So if you do a **read** after an **edit** command such as

```
e test
r test
```

the buffer will contain two copies of the original text as follows:

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

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

The **read** command (**r**) may also be used to read a file external to the buffer into the file in the buffer. This means you can append another file to the file you are editing. While in **ed** and at the current line, enter the command

```
.r filename
```

and *filename* will be read into the file (already in the buffer) immediately after the current line. A dot "." is a shorthand notation for the current line. None of the file in the buffer is destroyed, the external file *filename* has been read into and been combined with the file already in the buffer. The file that was read remains in *filename* also. You only copied it. Notice the difference between "**r**" and "**.r**". The **r** command appends another file at the end of the file being edited. The **.r** command appends another file after the current line.

Example Two

Experiment with the **e** command. Try reading and printing various files. You may get an error ?*name* where *name* is the name of a file. This means that the file does not exist. Some typical causes of getting an empty file are spelling the file name wrong or perhaps trying to read or write a particular file that your permissions will not allow. Try alternately reading and appending to see that they work similarly. Verify that the command

```
$ ed filename
system response
```

is exactly equivalent to

```
$ ed
e filename
system response
```

Printing Buffer Contents (p)

To print or list the contents of the buffer (or parts of it) on the terminal, use the **print** command (**p**). This is done as follows. Specify the line numbers where printing is to begin and end. These numbers have a comma between the beginning number and the ending number, that is, "beginning line number, ending line number p." For example, to print the first ten lines of the contents of any buffer (that is, lines 1 through 10), type

```
1,10p
```

The **ed** program will respond by printing the specified starting line (1) through the specified ending line (10).

To print all of the lines in the buffer, use **1,30p** as above if it is known that there are exactly 30 lines in the buffer. But in general, it is not known how many lines there are. To print all the lines in the buffer without knowing the exact number beforehand, the **ed** program uses the dollar sign (**$**) for the last number in the buffer. To print all the lines in the buffer, use either of the following command entries:

```
1,$p
,p
```

Either of these commands will print the lines in the buffer (line 1 through the last line). The **1,$p** entry can be abbreviated **,p**. To halt the printing, press the DELETE key. The **ed** program will respond

```
?
```

and wait for the next input command.

To print the last line of the buffer, you could use

```
$,$p
```

but **ed** lets you abbreviate this to

```
$p
```

Any single line can be printed by typing the line number followed by a p. Thus

```
1p
```

produces the line

```
Now is the time
```

that is the first line of the buffer.

The **ed** lets you abbreviate even further. You can print any single line by typing just the line number without any need to type the letter p. So if you enter

```
$
```

ed will print the last line of the buffer. Entering a single line number will print that line only.

It is also possible to use **$** in combinations like

```
$-5,$p
```

that prints the last five lines of the buffer. This helps to determine the end of the contents of the buffer when more is to be entered.

Example Three

Create some text using the **a** command and experiment with the **p** command. You will find, for example, that line 0 or a line beyond the end (last line) of the buffer cannot be printed. Attempts to print a buffer in reverse order by entering

```
3,1p
```

will not work.

Identifying the Current Line (.)

Suppose the buffer still contains the six lines of text (as in example one), and the following was entered

```
1,3p
```

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

```
p
```

This will print

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

that is the third line of the buffer. It is the last (most recent) line that anything was done to (in this example it is the line just printed). The **p** command can be repeated without line numbers, and it will continue to print line 3.

The **ed** maintains a record of the last line that anything was done to so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

```
.
```

Dot (.) is a line number in the same way that **/fB$** is. Dot means exactly "the current line," or loosely, "the line something was done to most recently." The dot can be used in several ways. One possibility is to enter

```
.,$p
```

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

Some commands change the value of dot, while others do not. The **print** command (**p**) sets dot (.) to the number of the last line printed; the last command entered (**.,$p**) will set dot to the last line in the buffer (line 6).

Dot is most useful when used in combinations like

```
.+1
```

(or equivalently, .+**1p**). This means "print the next line" and is a handy way to step slowly through a buffer. You can also enter

```
.-1
```

(or **.-1p**), that means "print the line before the current line." This enables stepping through the buffer backwards if desired. Another useful one is something like

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

that prints the previous three lines.

Do not forget that this changes the value of dot. You can find out what the line number of dot is at any time by typing

```
.= (dot line number is ?)
```

The **ed** program will respond by printing the value (line number) of dot.

Let us summarize some things about the **p** command and dot. The **p** can be preceded by no line numbers, a specific line number, or a range of line numbers. If there is no line number given, it prints the current line, the line that dot refers to. If there is one line number given with or without the letter **p**, it prints that line and sets dot there. If there are two line numbers separated by a comma, it prints all the lines in that range from the first number to the last number, and sets dot to the last line printed. If two line numbers are specified, the first cannot be bigger than the second (refer to the beginning of example three).

Pressing the RETURN key causes the next line to print. It is equivalent to

```
.+1p
```

Entering a circumflex (^) is equivalent to entering a hyphen (-). It can be used in multiples, as ^^^, that will move the current line or dot line backwards three lines from the current line. The "-" or the "^" can be considered equivalent to **-1p** since either moves the dot back one line.

Deleting Lines (d)

Suppose some lines in the buffer are not needed. They may be removed by use of successive delete commands

```
d
```

When **d** is entered, the deleted lines are not printed. A delete action is similar to that of the **print** command (**p**). The lines to be deleted are specified for **d** exactly as they are for **p** as follows:

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

Thus the command

```
4,$d
```

deletes lines 4 through the end. There are now three lines left. This can be checked by using

```
1,$p
```

Notice that **$** now is line 3. Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to **$**. The **delete** command (**d**) and the **print** command (**p**) may be used together. Thus

```
dp
```

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

Example Four

In trying out these commands, you will find that **a** will append lines after the line number that you specify (rather than after dot); **r** reads a file in after the line number you specify (not necessarily at the end of the buffer); and **w** will write out exactly the lines specified, not necessarily the whole buffer. These variations are sometimes handy. For instance, a file can be inserted at the beginning of a buffer by entering

```
0r filename
```

Lines can be entered at the beginning of the buffer by using

```
0a
text
.
```

Notice that **.w** is different from

```
.
w
```

Changing Text (s)

We are now ready to try another useful command, the **substitute** command

```
s
```

The **subsititute** command is used to change individual words or letters within a line or group of lines. It is used for correcting spelling mistakes and typing errors.

Suppose that, because of a typing error, line 1 says

```
Now is th time
```

Notice the e has been left off. The **s** command can be used to fix this as follows:

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

This says: in line 1, substitute for the characters "th" the characters "the". Since **ed** will not print the result automatically, enter

```
p
```

to verify that the substitution worked. You should get

```
Now is the time
```

Notice that dot was set to the line where the substitution took place since the **p** command printed that line. Dot is always set this way with the **s** command.

The general format of the **substitute** command is

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

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair in all the lines between the *starting line* and *ending line*. Only the first occurrence on each line is changed however. (If every occurrence is to be changed, see Example Five.)

The rules for line numbers are the same as those for the **print** command (**p**) except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is not changed. This causes an error response **?** as a warning.)

Thus the following can be entered

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

to correct the first spelling mistake (speling here) on each line in the text.

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

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

that makes some correction on the current line and then prints it (current line) to make sure it worked out right. If it did not, try again. Notice that there is a **p** on the same line as the **s** command. With few exceptions, **p** can follow any **substitute** command.

It is also legal to say

```
s/string//
```

that means change *string* to nothing. In other words, remove the characters. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if the buffer contained

```
Nowxx is the time
```

this can be corrected by entering

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

to get

```
Now is the time
```

Notice that // (two adjacent slashes) means "no characters", not a blank. (See "Context Searching" in Example Five for another meaning of "//").

Example Five

To experiment with the **substitute** command, see what happens if you substitute for some word on a line with several occurrences of the same word. For example, enter

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

This results in the following:

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

A **substitute** command changes only the first occurrence of the first string. You can change all occurrences by adding the **g** (for **global**) command to the **s** command. Change all occurrences of a string on a line by entering

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

Notice that slashes delimit the two sets of characters in the **s** command. Other characters can be used to delimit instead of slashes. You can use anything except blanks or tabs. For example, try using different delimiters like

```
s?...?...?gp
```

If the delimiter is the same as the character to be changed, you will have problems. For example, to change 'occcur' to 'occur' by entering

```
scccccc
```

will not work, but entering

```
saccac
```

will work.

Some characters have a special meaning to the editor when used within the substitute editing command. These characters are called metacharacters. Metacharacters also have a special meaning to shell commands. A few of the metacharacters are

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

Metacharacters are useful, however, avoid using metacharacters in substitutions until you are familiar with them. If you use metacharacters improperly, you may get strange results; but this can be undone by entering

```
up
```

before writing or doing another substitution. The **u** command undoes the last **substitute** command. For more details about metacharacters, read the subsection "Special Characters" in this section.

Context Searching (/...../)

After practicing the **substitute** command, move on to another feature-context searching.

If the original three lines of text in the buffer are as follows:

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

and the word "their" is to be changed to "the", how is the line that contains "their" located? With only three lines in the buffer, it is easy to keep track of what line the word "their" is on. But if the buffer contained several hundred lines and you had been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be. Context searching is a method of specifying the desired line, regardless of the number, by specifying some context (string).

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

```
/string/
```

For example, the **ed** expression

```
/their/
```

is a context search that finds the desired line. It will locate the next occurrence of the characters between the slashes. It also sets dot to that line and prints that line for verification as follows:

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

Searching for the next occurrence causes **ed** to start looking for the string at line .+**1** and search to the end of the buffer. If not found the search wraps around to line 1 and continues to the current line if necessary. That is, the search wraps around from **$** to **1**. It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters cannot be found in any line, **ed** types the error message

```
?
```

Otherwise, it prints the line it found.

The search for the desired line and the substitution can be done together, like this

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

This will yield

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

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

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

Suppose the buffer contains the three familiar lines

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

Then the **ed** line numbers

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and they all refer to the same line (line 2). To change something in line 2, enter

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

or

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

or

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

The choice is dictated only by convenience. All three lines could be printed by entering

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

or

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

or by many similar combinations. The first example would be better if you do not know how many lines are involved.

The basic rule is: a context search expression acts like a line number, so it can be used wherever a line number is needed.

Example Six

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

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

Try context searching using *?text?* instead of using */text/*. This scans lines in the buffer in reverse order instead of the normal forward order. This is sometimes useful if you go too far while looking for some string of characters and is an easy way to back up.

If unusual results are obtained with any of the characters

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

refer to the subsection "Special Characters."

The **ed** program also provides a short method for repeating a context search for the same string. For example, the **ed** line number

```
/string/
```

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

```
//
```

This method searches for the most recently (last) used context search expression. It can also be used as the first string of the **substitute** command, as in

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

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

```
??
```

will scan backwards for the same expression.

Change and Insert Commands (c and i)

The **change** command (**c**) is used to change or replace the specified lines with a group of one or more lines. The **insert** command (**i**) is used to insert a group of one or more lines immediately before the current line.

The **change** command, written as

```
c
```

is used to replace lines with different lines that are typed in at the terminal. For example, to change the first line (**.+1**) through the last line (**$**) of a file, enter

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

The lines typed between the **c** command and the "." (dot) command will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines that have errors.

If only one line is specified in the **c** command, then just that line is replaced. (As many replacement lines as you like can be added.) Notice the use of "." (dot) to end the input. This works just like the "." (dot) in the **a** command and must appear by itself at the beginning of a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line typed.

The **insert** command is similar to **append**, for instance

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

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

Example Seven

The **c** command is like a combination of **delete** followed by **insert**. Experiment to verify that

```
starting-line,ending-line d
i
text
.
```

is almost the same as

```
starting-line,ending-line c
text
.
```

These are not precisely the same if the last line (**$**) gets deleted.

Experiment with the **append** command (**a**) and the **insert** command (**i**) to see that they are similar but not the same. You will observe that

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

appends after the given line, while

```
line-number i
text
```

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

Moving Text Around-(m)

The **move** command (**m**) is used for cutting and pasting. It allows a group of lines to be moved from one place to another in the buffer. Suppose the first three lines of the buffer is to be placed at the end of the buffer instead of at the beginning. This could be performed by entering

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

This method will work, but it is easier to use the **m** command as follows:

```
1,3m$
```

The general format for the **move** command is
starting-line,ending-line **m** *after-this-line*

Notice that there is a third line number to be specified; the line number that the moved lines are to follow. Of course, the lines to be moved can be specified by context searches. If you had

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

the two paragraphs could be reversed like this:

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

Notice the **-1** (the moved text) goes the line mentioned. Dot gets set to the last line moved.

The Global Commands

The two **global** commands are **g** and **v**. The **global** command **g** is used to execute one or more **ed** commands on all those lines in the buffer that match some specified string. For example,

```
g/string/p
```

prints all lines that contain *string*. More usefully,

```
g/string1/s//string2/gp
```

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

```
1,$s/string1/string2/gp
```

that only prints the last line substituted. Another subtle difference is that the **g** command does not give a **?** if *string1* is not found, but the **s** command will.

There may be several commands used with the **g** command, but every line except the last must end 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 that contains the string "xxx," then prints all three lines.

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

```
v/ /d
```

deletes every line that does not contain a blank.

Special Characters

You may have noticed that context searches and substitutions do not work when used with metacharacters. Some of the metacharacters follow:

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

The **ed** program treats these characters as special, with special meanings. For instance, in a context search or the first string of the **substitute** command,

```
/x.y/
```

will search for a line with an x, any character. and a y. It does not search only for a line with an x, a period, and a y. The character "." matches anything. For example, the input

```
/x.y/
```

matches any of the following:

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

The following is a complete list of the special characters that can cause trouble:

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

Note: *The backslash character* (\) is special to the ***ed*** *program. Avoid using it when possible.*

If you have to use a special character in a **substitute** command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

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

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

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

```
/^string/
```

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

```
string
```

but not

```
the string
```

The dollar sign (**$**) is just the opposite of the circumflex. The dollar sign is used to search for *string* at the end of a line. The input

```
/string$/
```

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

```
/^string$/
```

will find a line that contains just *string*.

The dot (.) character is useful with the repetition character (*). The **a*** is a shorthand input that matches several a's. Therefore .* matches several anythings (including no everythings). For example, input

```
s/.*/string/
```

to change an entire line to *string*, or

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

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

```
/^.$/
```

to find a line containing exactly one character.

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

```
/[0123456789]/
```

matches any single digit. If any character inside the brace is found, a match occurs. This can be abbreviated to

```
[0-9]
```

Letters can also be used inside the brackets.

Finally, the ampersand (**&**) is another shorthand character. It is used only on the right-hand part of a **substitute** command where it means "whatever was matched on the left-hand side." It is used to save typing. Suppose the current line contained

```
Now is the time
```

and you wanted to put parentheses around it. One tedious method is just to retype the line. Another method is to enter

```
s/^/(/
```

or

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

^ and **$**. But the easiest way uses the **&** as follows:

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

This will match the whole line and replace it by itself surrounded by parentheses. The **&** can be used several times in a line. Try using

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

to produce

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

You do not 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
```

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

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

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

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

Summary of Commands

The general form of the **ed** text editor commands is the command name, perhaps preceded by one or two line numbers. For the **edit** command (**e**), the read command (**r**), and the **write** command (**w**), the command name is also followed by a file name.

a	Append adds lines to the buffer (at line dot, unless a different line is specified). Appending continues until a dot (.) is typed at the beginning (first character) of a new line. Dot is set to the last line appended.
c	Change the specified lines to the new text that follows. Entering new lines is ended by a dot (.) on a line by itself. If no lines are specified, the current line (dot) is replaced. Dot is set to last line changed.
d	Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless **$** is specified. When **$** is specified, dot is set to the last line (**$**.)
e	Edit new file. The **edit** command causes the entire contents of your present buffer to be deleted. After the buffer contents are thrown away, the named file is read in the buffer. This command is not covered in this section.
f	Print the remembered file name. If a name follows **f**, the remembered name will be set to it.
g	The global command **g**/*string*/*commands* will execute the *commands* on those lines that contain *string*.
i	Insert lines before the specified line or the current line (dot line) until a "." is typed at the beginning of a new line. Dot is set to last line inserted.
m	Move lines specified to the line named after **m**. Dot is set to the last line moved.

n	Print the line number addressed, followed by a tab and the line itself.
p	Print specified lines. If no lines are specified, print the current line (line dot). Pressing the RETURN key once prints the next line in your file.
q	The **quit** command exits from **ed**. It does not save changes, if you give it twice in a row without first giving a **write** command (**w**).
r	Read a file into buffer (at end unless specified elsewhere). Dot set to last line read. If **.r** *filename* is used, *filename* is read into the buffer immediately after the dot line.
s	The *s/string1/string2/* command is used to substitute *string1* with *string2* in the specified lines. If no lines are specified, the substitution is done to your current line. Dot is set to the last line in which a substitution took place. If no substitution took place, dot is not changed. The command **s** changes only the first occurrence of *string1* on a line. To change all occurrences on a line, type a **g** after the final slash.
v	The **exclude** command **v**/*string*/*commands* executes *commands* only on those lines that contain *string*.
w	The **write** command writes out the buffer contents onto a file. Dot is not changed.
.=	The **.=** causes the printout of the current line number. The dot value prints the current line number. The = by itself prints the value of the last line in the file.
!	The exclamation mark (!) is a **temporary escape** command. The line "command- line" causes "command-line" to be executed as a CENTIX system command.
/string/	This command performs a context search for the next line that contains this string of characters *"string"* and prints it. Dot is set to the line where string was found. Search starts at line **.=1** wraps around from the last line **$** to line 1 and continues to dot (the current line) if necessary.
?*string*?	This command performs a context search in the reverse direction. The search starts at the line before your current line, scans back to line 1, wraps around to the last line in your file, and then scans back to your current line, if necessary.

Text Editor (ed)--Avanced

The advanced editing part is meant to help CENTIX operating system users (secretaries, typists, programmers, and so on) make effective use of the more complex capabilities of **ed**. It provides explanations and examples of

- Special characters, operating on lines, line addressing, and global commands in **ed**.
- Commands for operations on files and parts of files, including **r**, **w**, **m**, and **t** commands of **ed**.

Examples are based on experience and observations of users and the difficulties they have encountered.

The CENTIX operating system provides tools for text editing, but that by itself is no guarantee that everyone will make the most effective use of them. In particular, users who are not computer specialists (typists, secretaries, casual users) often use the CENTIX operating system less effectively than they could. Although this document is written for nonprogrammers, CENTIX users, regardless of their background, should find helpful hints on how to get their jobs done more easily.

The next paragraphs discuss shortcuts and labor-saving devices. Not all will be instantly useful and others should provide ideas for future use.

Special Characters

The **ed** program is the primary interface to the system, so it is worthwhile to know how to get the most out of it.

Print and List Commands

Two commands are provided for printing contents of lines being edited. Most users are familiar with the **print** command (**p**) in combinations like

```
1,$p
```

to print all lines that are being edited, or

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

to change the first abc to def on the current line and to print the results. Less familiar is the **list** command (**l**) which gives slightly more information than **p**. In particular, **l** makes characters visible that are normally invisible, such as tabs and backspaces. If a line contains some of these, the **l** command will print each tab as ">" and each backspace as "<". This makes it easier to correct typing mistakes that insert extra spaces adjacent to tabs or a backspace followed by a space.

The **l** command also "folds" long lines for display purposes. Any line that exceeds 72 characters is printed on multiple lines. Each printed line except the last is automatically terminated by a backslash (\) to indicate that the line was folded. A dollar character (**$**) is appended to the real end of line. This is useful for printing long lines on terminals having output line capability of only 72 characters per line.

Occasionally, the **l** command will print in a line a string of numbers preceded by a backslash, such as **\07** or **\16**. These combinations are used to make characters visible that normally do not print, for example, formfeed, vertical tab, or bell. Each such combination is interpreted as a single character. When such characters are detected, they may have surprising meanings when printed on some terminals. Often their presence means that a finger slipped while typing.

Substitute Commands

The **substitute** command (**s**) is used for changing the contents of individual lines. It is probably the most complex and effective of any **ed** command.

The meaning of a trailing **global** command (**g**) after a **substitute** command is illustrated in the next two commands:

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

and

```
s/abc/def/g
```

The first form replaces the first abc on the line with def. If there is more than one occurrence of abc on the line, the second form (with the trailing **g**) changes all of them. Either of the two forms of the **s** command can be followed by **p** or **l** to print or list the contents of the line:

```
s/abc/def/p
s/abc/def/l
s/abc/def/gp
s/abc/def/gl
```

All are legal and have slightly different meanings.

An **s** command can be preceded by one or two line numbers to specify that the substitution is to take place on a group of lines. Thus:

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

changes the first occurrence of "mispell" to "misspell" on every line of the file. The following command changes every occurrence on every line:

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

By adding a **p** or **l** to the end of any of these **substitute** commands, only the last line that was changed will be printed, not all lines. How to print all the lines that were changed is described in a later paragraph.

Any character can be used to delimit pieces of an **s** command. There is nothing sacred about slashes (but slashes must be used for context searching). For instance, for a line that contains a lot of slashes already, for example:

```
//exec //sys.fort.go // and so on...
```

a colon could be used as the delimiter. To delete all the slashes, the command is

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

Undo Command

Occasionally, an erroneous substitution is made in a line. The **undo** command (**u**) negates the last command so that data is restored to its previous state. This command is especially useful after executing a **global** command if it is discovered the command did things that are undesirable.

Metacharacters

When using **ed**, certain characters have special meanings when they occur in the left side of a **substitute** command or in a search for a particular line. These special characters are called "metacharacters" and are listed below:

- Period (.)
- Backslash (\)
- Dollar Sign ($)
- Circumflex (^)
- Star (*)
- Brackets ([ ])
- Ampersand. (&)

Even though metacharacters are discussed separately in the following text, they can be combined.

Period

The period (.) on the left side of a **substitute** command or in a search stands for any single character. Thus the search

```
/x.y/
```

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

```
x+y
x-y
x<sp>y
x.y
```

The <**sp**> stands for a space character whenever needed to make it visible.

Since the period matches any single character, a way to deal with the "invisible" characters printed by **l** is available. For instance, if there is a line that when printed with the **l** command, appears as

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

and it is desired to get rid of the \07 (the bell character), the most obvious solution is to try

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

However, this will fail. Retyping the entire line is a reasonable tactic if the line in question is not too long. However, for a very long line, retyping could result in additional errors. Since \07 really represents a single character, the command

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

gets the job done. The period matches the mysterious character between the "h" and the "i".

Since the period matches any single character, the command

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

converts the first character on the line into a ",".

As is true of many characters in **ed**, the period has several meanings depending on its context. In the line

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

- The first period is the line number of the line being edited, which is called "dot."
- The second period is a metacharacter that matches any single character on that line (in this instance the first character of the line).
- The third period is the only one that really is a literal period. On the right side of a substitution, the period is not special.

Backslash

Since a period means "any character," the question arises of what to do when a period is really needed. For example, to convert the line:

Now is the time.

into

Now is the time?

the backslash (\) is used. A backslash turns off any special meaning that the next character might have. In particular, \. converts the period from a "match anything" into a "match the period" statement. The \. pair of characters is considered by **ed** to be a single literal period. To replace the period with a question mark, the following command is used:

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

The backslash can also be used when searching for lines that contain a special character. If a search is made to look for a line that contains

```
.PP
```

the search

```
/.PP/
```

is not adequate. It will find a line like

THE APPLICATION OF ...

The period matches the letter "A". But if the command

```
/\.PP/
```

is used, only the lines that contain ".PP" are found.

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

```
/\/
```

will not work because the \ is not a literal backslash, but instead means that the second / no longer delimits the search. A search can be made for a literal backslash by preceding a backslash with another \:

```
/\\/
```

Similarly, searches can be made for a slash (/) with

```
/\//
```

The backslash turns off the meaning of the immediately following / so that it does not terminate the search prematurely.

Some **substitute** commands, each of which will convert the line

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

into the line

```
\x\y
```

are

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

The user's erase character and the line kill character (# and @ by default) must also be used with a backslash to turn off their special meaning. This is a feature of the CENTIX operating system. When adding text with **append (a)**, **insert (i)**, or **change (c)** commands, the backslash is special only for the erase and line kill characters; and only one backslash should be used for each one needed.

Dollar Sign

In the left side of a substitute command or in a search command, the dollar sign (**$**) stands for "the end of line." The word "time" is added to the end of the following phrase:

    Now is the

with the following command:

```
s/$/<sp>time/
```

The result is

    Now is the time

A space is needed before "time" in the **substitute** command; otherwise, the following will be printed:

    Now is thetime

The second comma in the following line can be replaced with a period without altering the first:

    Now is the time, for all good men,

The needed command is

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

The **$** provides context to indicate which specific comma. Without it, the **s** command would operate on the first comma to produce

Now is the time. for all good men,

To convert

Now is the time.

into

Now is the time?

that was previously done with the backslash, the following command is used:

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

The **$** has multiple meanings depending on context. In the line

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

- The first **$** refers to the last line of the file.
- The second **$** refers to the end of the line.
- The third **$** is a literal dollar sign to be added to that line.

Circumflex

The circumflex (^) stands for the beginning of the line. For example, if a search is made for a line that begins with "the," the command

```
/the/
```

will in all likelihood find several lines that contain "the" before arriving at the line that was wanted. But the command

```
/^the/
```

narrows the context, and thus arrives at the desired line more quickly.

The other use of the circumflex is to enable text to be inserted at the beginning of a line. For example:

```
s/^/<sp>/
```

places a space at the beginning of the current line.

Metacharacters also can be combined. For example, to search for a line that contains only the characters

```
.PP
```

the command

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

can be used.

Star

The star (*) is useful to replace all spaces between x and y with a single space, as in the following example:

```
text x    y text
```

where *text* stands for lots of text, and there are an indeterminate number of spaces between x and y. The line is too long to retype, and there are too many spaces to count.

A regular expression (typically a single character) followed by a star stands for as many consecutive occurrences of that regular expression as possible. To refer to all the spaces at once, the following command is used:

```
s/x<sp>*y/x<sp>y/
```

The construction **<sp>*** means "as many spaces as possible." Thus **x<sp>*y** means: "an x, followed by as many spaces as possible, and then a y."

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

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

then all - characters can be replaced by a single space with the command

```
s/x-*y/x<sp>y/
```

If the original line was

```
text x.........y text
```

and if the following command was typed:

```
s/x.*y/x<sp>y/
```

what happens depends upon the occurrence of other x's or y's on the line. If there are no other x's or y's, then everything works, but it is blind luck, not good management. Since a period matches any single character, then .* matches as many single characters as possible. Unless the user is careful the star can eat up a lot more of the line than expected. If the line was

```
text x text x.........y text y text
```

then the command will take everything from the first x to the last y, which, in this example, is undoubtedly more than is wanted. The proper way is to turn off the special meaning of period with \. as shown in the following command:

```
s/x\.*y/x<sp>y/
```

Now everything works since \.* means "as many periods as possible."

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

    Now is the time for all good men ...

into

    Now is the time.

the following deletes everything after the word "time":

```
s/<sp>for.*/./
```

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

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

and the command is

```
s/x<sp>*y/x<sp>y/
```

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

```
/x<sp><sp>*y/
```

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

The other behavior of * is also related to the zero being a legitimate number of occurrences of something followed by a star. The command

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

when applied to the line

```
abcdef
```

produces

```
yaybycydyeyfy
```

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there is no x at the beginning of the line (so that gets converted into a y), nor between the a and the b (so that gets converted into a y), and so on. The following command:

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

where **xx*** is "one or more x's", when applied to the line

```
abcdefxghi
```

produces

```
abcdefyghi
```

Brackets

Should a number that appears at the beginning of all lines of a file need to be deleted, a first thought might be to perform a series of commands like:

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

This is going to take forever if the numbers are long. Unless it is desired to repeat the commands over and over until finally all numbers are gone, the digits can be deleted on one pass. This is the purpose of brackets ([ ]).

The construction

```
[0123456789]
```

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

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

deletes all digits from the beginning of all lines.

Any characters can appear within a character class; and just to confuse the issue, there are essentially no special characters inside the brackets. Even the backslash does not have a special meaning. The following command searches for special characters within the brackets:

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

Within a character class, the [is not special. To get a] into a character class, it should be placed as the first character in the class. For example:

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

It is a nuisance to have to spell out the digits. They can be abbreviated as [0-9]; similarly, [a-z] stands for the lowercase letters and [A-Z] for uppercase letters.

The user can specify a character class that means "none of the following characters." This is done by beginning the class with a circumflex.

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

which stands for "any character except a digit." The following search finds the first line that does not begin with a tab or space:

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

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. For example:

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

finds a line that does not begin with a circumflex.

Ampersand

The ampersand (&) is used primarily to save typing. For example, if the following is the original line:

    Now is the time

and it needs to be

    Now is the best time

the command

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

can be used, but it is unnecessary to repeat the "the". The & is used to eliminate the repetition. On the right-hand side of a **substitute** command, the ampersand means "whatever was just matched," so in the command

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

the & represents "the". This is not much of a saving if the thing matched is just "the"; but if it is something long or complicated or if it is something (such as .*) which matches a lot of text, the & can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line regardless of its length:

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

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

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

makes the original line

Now is the best and the worst time

and

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

converts the original line into

Now is the time? Now is the time!!

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

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

converts the word into the symbol. The **&** is not special on the left-hand side of a substitute, only on the right.

Operating on Lines

Substituting Newline Characters

The **ed** program provides a facility for splitting a single line into two or more lines by substituting in a newline character. If a line is unmanageably long because of editing or merely because of the way it is typed, it can be divided as follows:

```
text xy text
```

can be broken between the x and the y with the following **substitute** command:

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

This is actually a single command although it is typed on two lines. Since the \ turns off special meanings, a \ at the end of a line makes the newline character no longer special. When a new line is substituted, dot is left pointing at the last line created.

A single line can be made into several lines with this same mechanism. The word "very" in the following example can be put on a separate line preceded with the **nroff** formatter **underline** command (**.ul**):

```
text a very big text
```

The commands

```
s/<sp>very<sp>/\
.ul\
very\
/
```

convert the line into four shorter lines

```
text a
.ul
very
big text
```

The word "very" is preceded by the line containing the **.ul** and spaces around "very" are eliminated at the same time.

Joining Lines

Lines may be connected with the **join** command (**j**). Given the lines

```
Now is
<sp>the time
```

if dot is set to the first line, then the **j** command joins them together. No spaces are added, which is why a space is shown at the beginning of the second line.

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

```
1,$jp
```

joins all the lines into a big one and prints it.

Rearranging Lines

The **&** metacharacter stands for whatever was matched by the left side of an **s** command. Similarly, several pieces can be captured of what was matched; the only difference is it must be specified on the left side just what pieces the user is interested in. For instance, if there is a file of lines that consist of names in the form

Smith, A. B.

Jones, C.

and so on and it was intended to have the initials to precede the name, as in:

A. B. Smith

C. Jones

it is possible to rectify this with a series of tedious and error-prone editing commands. The alternative is to "tag" the pieces of the pattern (in this case, the last name and the initials) and then rearrange the pieces. On the left side of a substitution if part of the pattern is enclosed between \(and \), whatever matched that part is remembered and available for use on the right side. On the right side, the symbol **\1** refers to whatever matched the first \(...\) pair, **\2** to the second \(...\) pair, and so on.

The command

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

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

With any complicated editing sequence, it is foolhardy to run it and hope. **Global** commands provide a way to print those lines affected by the **substitute** command.

Line Addressing in the Editor

Line addressing in ed specifies the lines to be affected by editing commands. Previous constructions like

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

were used to specify a change on all lines. Most users are familiar with using a single newline character (or return) to print the next line and with

```
/string/
```

to find a line that contains *string*. Less familiar is the use of

```
?string?
```

to scan backwards for the previous occurrence of *string*. This is handy when the user realizes that the string to be operated on is back up the page (file) from the current line being edited. The slash and question mark are the only characters that can be used to delimit a context search.

Address Arithmetic

It is possible to combine line numbers like ., **$**, /.../, and **?...?** with + and -. Thus:

```
$-1
```

is a command to print the next to last line of the current file (that is, one line before line **$**). For example:

```
$-5,$p
```

prints the last six lines. If there are not six lines, an error message will be indicated.

As another example:

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

prints from three lines before the current line to three lines after. The + can be omitted

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

is identical in meaning.

Another area in which to save typing effort in specifying lines is by using - and + as line numbers by themselves. For instance, a single minus sign

```
-
```

by itself is a command to move back up one line in the file. Several minus signs can be strung together to move back up that many lines:

```
---
```

moves up three lines, as does **-3**. Thus:

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

is also identical to the examples above.

Since - is shorter than **.-1**, constructions like

```
-,.s/bad/good/
```

are useful. This changes the first occurrence of "bad" to "good" on both the previous line and the current line.

The + and - can be used in combination with searches using */string/*, *?string?*, and **$**. The search

```
/string/--
```

finds the line containing *string* and positions dot two lines before it.

Repeated Searches

When the search command is

```
/horrible string/
```

and when the line is printed, it is discovered that it is not the *horrible string* that was wanted. It is necessary to repeat the search again, but it is not necessary to retype it. The construction

```
//
```

is a shorthand for "the string that was previously searched for," whatever it was. This can be repeated as many times as necessary. This also applies to the backwards search

```
??
```

which searches for the same string but in the reverse direction.

Not only can the search be repeated, but the // construction can be used on the left side of a **substitute** command to mean "the most recent pattern":

```
/horrible string/
```

ed prints line with *horrible string*.

```
s//good/p
```

ed prints the corrected line.

To search backwards and change a line, the following command is used:

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

Of course, the **&** on the right-hand side of a substitute can still be used to stand for whatever got matched

```
//s//&<sp>&/p
```

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

Default Line Numbers

One of the most effective ways to speed editing is by knowing which lines are affected by a command with no address and where dot will be positioned when a command finishes. Editing without specifying unnecessary line numbers can save a lot of typing. As the most obvious example, the **search** command

```
/string/
```

puts dot at the next line that contains *string*. No address is required with commands like:

- **p** to print the current line
- **l** to list the current line
- **d** to delete the current line

- **a** to append text after the current line
- **c** to change the current line
- **i** to insert text before the current line
- **s** to make a substitution on the current line.

If there was no *string* detected, dot stays on the line where it was. This is also true if it was sitting on the only *string* when the command was issued. The same rules hold for searches that use ?*string*?; the only difference is direction of search.

The **delete** command (**d**) leaves dot at the line following the last deleted line. However, dot points to the new last line when the last line is deleted.

Line-changing commands **a**, **c**, and **i** affect (by default) the current line if no line number is specified. They behave identically after appending, changing, or inserting dot points at the last line entered. For example, the following can be done without specifying any line number for the **substitute** command or for the second **append** command:

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

The following overwrites the major error and permits continuation of entering information:

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

The **read** command (**r**) will read a file into the text being edited, either at the end if no address is given or after the specified line if an address is given. In either case, dot points at the last line read in. The **0r** command can be used to read in a file at the beginning of the text and the **0a** or **1i** commands can be used to start adding text at the beginning.

The **write** command (**w**) writes out the entire file. If the command is preceded by one line number, that line is written. Preceding the command by two line numbers causes a range of lines to be written. The **w** command does not change dot, therefore, the current line remains the same regardless of what lines are written. This is true even if a command like

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

is made which involves a context search. Since the **w** command is easy to use, the text being edited should be saved regularly just in case the system crashes or a file being edited is clobbered.

The command with the least intuitive behavior is the **s** command. The dot remains at the last line that was changed. If there were no changes, then dot is unchanged. To illustrate, if there are three lines in the buffer and dot is sitting on the middle one

```
x1
x2
x3
```

the command

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

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

```
x1
y2
y3
```

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

Semicolon

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

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

Starting at line 1, one would expect that the command

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

prints all the lines from the ab to the bc, inclusive. This is not what happens. Both searches (for a and for b) start from the same point, and thus they both find the line that contains ab. The result is to print a single line. If there had been a line with a b in it before the ab line, then the **print** command would be in error since the second line number would be less than the first; and it is illegal to try to print lines in reverse order. This is because the comma separator for line numbers does not set dot while each address is processed. Each search starts from the same place.

In **ed**, the semicolon (;) can be used just like comma with the single difference that use of a semicolon forces dot to be set at that point while line numbers are being evaluated. In effect, the semicolon moves dot. Thus in the example above, the command

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

prints the range of lines from ab to bc because after the a is found, dot is set to that line, and then b is searched for starting beyond that line. This property is most often useful in a very simple situation. If the need is to find the second occurrence of *string*, then the commands

```
/string/
//
```

print the first occurrence as well as the second. The command

```
/string/;//
```

finds the first occurrence of *string* and sets dot there. Then it finds the second occurrence and prints only that line.

Searching for the second previous occurrence of *string*, as in

```
?string?;??
```

is similar. Printing the third, fourth, and so on occurrence in either direction is left as an exercise.

When searching for the first occurrence of a character string in a file where dot is positioned at an arbitrary place within the file, the command

```
1;/string/
```

will fail if *string* is on line 1. You can use the command

```
0;/string/
```

(one of the few places where 0 is a legal line number) to start the search at line 1.

Interrupting the Editor

If the user interrupts **ed** while performing a command by pressing the DELETE key, the file is put back together again. The file state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable. If the file is being read from or written into, substitutions are being made, or lines are being deleted, these will be stopped in some clean but unpredictable state in the middle of the command execution (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus if a user interrupts **ed** while some printing is being done, dot is not sitting on the last printed line or even near it. Dot is returned to where it was when the **p** command was started.

Global Commands

Basic

Global commands (**g** and **v**) are used to perform one or more editing commands on all lines of a file. The **g** command operates on those lines that contain a specified string. As the simplest example, the command

```
g/string/p
```

prints all lines that contain *string*. The string that goes between the slashes can be anything that could be used in a line search or in a **substitute** command; exactly the same rules and limitations apply. As another example:

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

prints all lines that begin with period.

The **v** command (there is no mnemonic significance to the letter "v") is identical to **g** except that it operates on those lines that do not contain an occurrence of the string. So

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

prints the lines that do not begin with period.

The command that follows **g** or **v** can be almost any command. For example:

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

deletes all lines that begin with period, and

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

deletes all empty (blank) lines.

Probably the most useful command that can follow a **global** command is the **substitute** command since this can be used to make a change and print each affected line for verification. For example, to change the word "This" to "THIS" everywhere in a file and verify that it worked, the command is

```
g/This/s//THIS/gp
```

The use of // in the **substitute** command means "the previous pattern", in this case, "This". The **p** command is done on every line that matches the pattern, not just those on which a substitution took place.

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

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

prints the line that follows each **.PP** macro (the signal for a new paragraph in some formatting packages). The + means "one line past dot," and

g/*string*/?^\.SH?1

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

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

prints all the lines that lie between lines beginning with the **.EQ** and **.EN** formatting macros.

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

Multiline

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

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

is sufficient. The backslash signals the **g** command that the set of commands continues on the next line. It terminates on the first line that does not end with \. A **substitute** command cannot be used to insert a newline character within a **g** command.

The command

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

does not work as expected. The remembered pattern is the last pattern that was actually executed, so sometimes it will be x (as expected) and sometimes it will be a (not expected). The desired pattern should be spelled out

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

It is also possible to execute **a**, **c**, and **i** commands (**append**, **change**, and **insert**) under a **global** command. As with other multiline constructions, all that is needed is to add a \ at the end of each line except the last. Thus to add a **.nf** and **.sp** command before each **.EQ** line, the following is typed:

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

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

It is good practice after each **global** command to check that the command did only what was desired. Surprises sometimes happen. When they do occur, the **undo** command (**u**) is useful to negate what was done by the last command.

File Manipulation

Text Editor Functions

The text editor (**ed**) function performs the following operations:

- Inserts one file into another.
- Writes out part of a file.
- Moves lines around.
- Copies lines.
- Marks.

File Names

It is important to know the editor (**ed**) commands for reading and writing files. Equally useful is the **edit** command (**e**). Within **ed**, the command

```
e newfile
```

says "edit *newfile* without leaving the text editor." The **e** command discards whatever is being worked on and starts over on *newfile*. This is the same as if one had quit with the **q** command and reentered **ed** with a new file name except that if a pattern has been remembered a command like // will still work.

When entering **ed** with the command

```
     ed file
```

ed remembers the name of the *file*, and any subsequent **e**, **r**, or **w** commands that do not contain a file name will refer to this remembered file. Thus:

```
     ed file1
          ---      (editing)
     w        (writes back in file1)
     e file2  (edit different file without leaving ed)
          ---      (editing on file2)
     w        (writes back in file2)
```

and so on does a series of edits on various files without leaving **ed** and without typing the name of any file more than once. By examining the sequence of commands in this example, it can be seen why many operating systems use **e** as a synonym for **ed**.

The current file name can be found at any time with the **f** command by typing **f** without a file name. Also, the name of a remembered file can be changed with **f**. A useful sequence is

```
     ed file1
     f file2
          ---     (editing)
```

This obtains a copy of file1 and guarantees that a subsequent **w** command without a file name will write to file2 and will not overwrite the original file.

Insert One File into Another

When a file is to be inserted into another, the **r** command can be used. For example, if the file "table" is to be inserted just after the reference to "Table 1", the following can be used:

```
     /Table 1/
     Table 1 shows that...     (response from ed)
     .r table
```

The critical line is the last one. The **.r** command reads a file in after dot. An **r** command without any address adds lines to the end of the file, so it is equivalent to the **$r** command.

Write Out Part of a File

Another feature is writing to another file part of the document that is being edited. For example, it is possible to split into a separate file the table from the previous example, so it can be formatted or tested separately. If in the file being edited, there is

```
         --- text
.TS
         --- data for table
.TE
         --- text
```

to isolate the table information in a separate file called "table," first the start of the table (the .TS line) is found, and then the interesting part is written on file table

```
/^\.TS/
.TS              (response from ed)
.,/^\.TE/w table
```

The same job can be accomplished with the single command

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

The point is that the **w** command can write out a group of lines instead of the whole file. A single line can be written by using one line number instead of two. For example, if a complicated line was just typed and it will be needed again, it should be saved and read in later rather than retyped:

```
a
         --- lots of stuff
         --- stuff to repeat
.
.w temp
a
         --- more stuff
.
.r temp
a
         --- more stuff
.
```

Move Lines Around

Moving a paragraph from its present position in a paper to the end can be done several ways. For example, it is assumed that each paragraph in the paper begins with the **.PP** formatting macro. Another method is to write the paragraph onto a temporary file, delete it from its current position, and then read in the temporary file at the end. If dot is at the **.PP** macro that begins the paragraph, this is the sequence of commands:

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

This states that from where dot is now until one line before the next **.PP** write onto file temp. The same lines are deleted and the file temp is read in at the end of the working file.

An easier way is to use the **move** command (**m**) that **ed** provides. This does the whole set of operations at one time without a temporary file. The **m** command is like many other **ed** commands in that it takes up to two line numbers in front to tell which lines are to be affected. It is also followed by a line number that tells where the lines are to go. Thus:

```
line1,line2m line3
```

says "move all the lines from line1 through line2 to after line3". Any of "line1", and so on, can be line numbers, strings between slashes or dollar signs, or other line specifications. If dot is at the first line of the paragraph, the command

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

will also accomplish this task.

As another example of a frequent operation, the order of two adjacent lines can be reversed by moving the first one after the second. If dot is positioned at the first line, then

```
m+
```

does it. It says to move the line to after the dot. If dot is positioned on the second line

```
m--
```

does the interchange.

The **m** command is more concise and direct than writing, deleting, and rereading. The main difficulty with the **m** command is that if patterns are used to specify both the line being moved and the target line, they must be specified properly or the wrong lines may be moved. The result of a botched **m** command can be a costly mistake. Doing the job a step at a time makes it easier to verify that each step accomplished what was wanted. It is also a good idea to issue a **w** command before doing anything complicated; then if an error is made, it is easy to back up.

Copy Lines

The **ed** program provides a **transfer** command (**t**) for making a copy of a group of one or more lines at any point. This is often easier than writing and reading. The **t** command is identical to the **m** command except instead of moving lines it duplicates them at the place referenced. Thus:

```
1,$t$
```

duplicates the entire contents that is being edited. A more common use for **t** is creating a series of lines that differ only slightly. For example:

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

Marking a Line

The **ed** program provides for marking a line with a particular name so that the line can be referenced later by its name regardless of its line number. This can be useful for moving lines and for keeping track of them as they move. The **mark** command is **k**. The mark name must be a single lowercase letter. The command

```
kx
```

marks the current line with the name x. If a line number precedes the **k**, that line is marked. The marked line can then be referred to with the address

```
'x
```

Marks are most useful for moving things around. The first line of the block to be moved is found and marked with **ka**. Then the last line of the block is found and marked with **kb**. Dot is then positioned at the place where the lines are to go and the following command is performed:

```
'a,'bm
```

Note: *Only one line can have a particular mark name associated with it at any given time.*

Temporary Escape

Sometimes it is convenient to temporarily escape from the text editor to do some CENTIX operating system command without leaving the text editor. The **escape** command (!) provides a way to do this. If the command

```
!any CENTIX operating system command
```

is entered, the current editing state is suspended; and the command asked for is executed. When the command finishes, **ed** will return a signal by printing another ! and editing can be resumed.

Any CENTIX operating system command may be performed including another **ed** (this is quite common). In this case, another ! can be done.

Editing Scripts

If a fairly complicated set of editing operations is to be performed on an entire set of files, the easiest thing to do is to make a script file (that is, a file that contains the operations to be performed), and then apply this script to each file in turn. For example, if every instance of "This" needs to be changed to "THIS" and every instance of "That" needs to be changed to "THAT" in a large number of files, a file script is made with the following contents:

```
g/This/s//THIS/g
g/That/s//THAT/g
w
q
```

The following is done:

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

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

By using the CENTIX operating system command interpreter **[sh(1)]**, a set of files can be cycled automatically with varying degrees of ease.

Visual Editor (vi)--Basic

This chapter provides an introduction to the visual editor (**vi**). **Vi** is a display-oriented, interactive text editor. When using **vi**, the screen of your terminal acts as a window for looking into the file being edited. Changes made to the file are reflected on the screen.

Using **vi**, you can insert new text into the file at any place quite easily. Most of the **vi** commands move the cursor around in the file. There are commands to move the cursor forward and backward in units of characters, words, sentences, and paragraphs. A small set of operators (such as **d** for delete and **c** for change) are combined with the motion commands to form operations, such as "delete a word" or "change a paragraph," in a simple and natural way. This regularity and the mnemonic assignment of commands to keys makes the editor command set easy to remember and use.

This section contains 13 extensive examples that will be useful in learning to use **vi**. An abbreviated example at the end of this section will help those interested in using **vi** for limited applications.

Special Purpose Keys

Special purpose keys are used by **vi** and because of their importance they will be covered early in this chapter. Their descriptions are as follows:

GO	When you are in the editor, hitting the GO key several times causes the editor to ring the bell to indicate that it is in an inactive state. Partially formed commands are canceled with the GO key. When you insert text in the file, text insertion is ended with the GO key.
RETURN	The RETURN key is used to signal the end of a line and to start execution of certain commands.
DELETE	The DELETE key generates an interrupt that tells the editor to stop what it is doing. This is a forceful way of making the editor return to the inactive state.
CODE	The CODE key is used with a number of keys to perform various functions. In this guide, the notation CODE-**d** means to press the **d** key while pressing down the CODE key.

Getting In and Out of vi

Setting Up Your Terminal Configuration

Vi will work on a large number of video display terminals, and new terminal types can be added to a terminal description file. So, before you can start using **vi**, you must tell the system the type terminal you are using. Consult with your system administrator to determine the code for your terminal. If your terminal does not have a code, one can be assigned and a description for the terminal created.

After logging on to your CENTIX system, enter the following command to identify a PT 1500 terminal:

```
$ TERM=pt
$ export TERM
```

Invoking the Visual Editor

After telling the system the kind of terminal you have, you can run **vi** on a file with the command

```
$ vi filename
```

The name of the file to edit is *filename*. After pressing the RETURN key, the screen should clear and the text of your file appear on the screen. If something else happens, perhaps you gave the system an incorrect terminal type code, in which case the editor may make a mess out of your screen. This happens when **vi** sends control codes for one kind of terminal to some other kind of terminal. In this case, you can type the colon and the q keys (**:q**) and then press the RETURN key. This should get you back to the CENTIX system shell. Determine what you did wrong (ask someone else if necessary) and try again.

Another thing that can go wrong is that you typed the wrong file name and the editor printed the following:

```
[]
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"filename" [New file]
```

If this happens, follow the above procedure for getting out of the editor. Then try again, spelling the file name correctly. Also, you may have typed a terminal code that is not recognized by the system. In this case, the following will be displayed.

```
$ vi filename
:Unknown terminal type
[using open mode]
"filename" x lines, y characters
```

Again follow the above procedure for getting out of the editor.

If the editor does not seem to respond to the commands you enter, an interrupt can be sent to the editor by pressing the DELETE key. You should now be able to leave the editor by entering the **:q** command.

Editor Copy in Buffer

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

Writing the Buffer

After working with the editor for a while, you may wish to write into the buffer as follows:

```
:w
```

Your changes will be written into the buffer.

Quitting the Visual Editor

If you wish to leave the editor instead of just writing, enter

```
:wq
```

This writes the contents of the editor buffer (including changes) back into the file you are editing and then quits the editor. You can also use the **ZZ** command to write and leave the editor. If you want to end an editing session and discard all changes, the following can be entered:

```
:q!
```

Be very careful not to use this command when you really want to save the changes made.

Example One

I Accessing, writing, and quitting a file.

1 Make a copy of a familiar file.

2 To access the file copy, enter the **vi** command and *filename* followed by pressing the RETURN key.

```
$ vi filename
```

2 The file name will appear in quotes at the bottom of the screen. The number of lines and the number of characters will appear momentarily next to the file name. The first part of the file will then appear on the screen. The cursor will be at the top left character in the file. The file name, number of lines, and number of characters will remain displayed on the bottom line of the screen.

3 Enter the colon command symbol.

```
:
```

The colon appears below the bottom line of the displayed file displacing the file name and the number of lines and characters.

4 Enter the **write** command.

```
:w
```

The **w** appears at the bottom of the screen next to the colon.

5 Press the RETURN key.

The file name in quotes, the number of lines, and the number of characters appear at the bottom of the screen.

6 Enter the **quit** command.

```
:q
```

A q will appear next to the colon at the bottom of the screen. After pressing the RETURN, the system prompt is displayed on the next line at the bottom of the screen. You have exited the **vi** editor. No change has occurred in your file. (The above procedure can be accomplished by entering **:wq**).

II Accessing and quitting a file after writing has occurred.

1 Access your file for the second time.

```
$ vi filename
```

Wait for a screen display. You could enter information at this time. In this exercise, no information will be entered.

2 Enter the **write** command.

```
:w
```

The **write** command appears at the bottom of the screen. After pressing the RETURN key, the file name in quotes, the number of lines, and the number of characters appear at the bottom of the screen.

3 Enter the **quit** command.

```
:q
```

The file name, number of lines, and number of characters is replaced by the :. A q appears at the bottom of the screen next to the colon. After pressing the RETURN key, the system prompt appears at the bottom of the screen. You have left the **vi** editor once again. If changes had been made in your buffer file, they would have been written into your original file.

III Accessing and quitting a file after writing has occurred--a second method.

1 Access your file.

```
$ vi filename
```

Wait for a screen display. You could enter information at this time. In this example no information will be entered.

2 Enter a write and quit.

```
:wq
```

A **wq** appears at the bottom of the screen next to the colon. After pressing the RETURN key, the file name, number of lines, and number of characters appear at the bottom of the screen. The system prompt appears on the next line. You have left the **vi** editor. If changes had been made in your buffer file, they would have been written into your original file.

IV Accessing and quitting a file after writing has occurred--a third method.

1 Access your file.

```
$ vi filename
```

Wait for a screen display. You could enter information at this time. In this example no information will be entered.

2 Enter the **write** and **quit** command in uppercase as follows:

```
ZZ
```

The system prompt appears at the bottom of the screen. You have left **vi** editor again. If changes had been made in your buffer file, they would have been written into your original file.

The **ZZ** command is identical to the **:wq** command, except that it does not respond with the file name, number of lines, and number of characters when no changes have been made.

V Accessing and quitting a file without saving changes.

1 Access your file.

```
$ vi filename
```

Wait for a screen display. You could enter information at this time. In this exercise, no information will be entered.

2 Enter the quit command without saving changes command.

```
:q!
```

The three characters appear at the bottom of the screen. After pressing the RETURN key, the system prompt appears at the bottom of the screen. If any changes had been made in your buffer file, they would not be saved.

VI Changing a file name.

1 Access your file.

```
$ vi filename
```

Wait for a screen display. You could enter information at this time. In this example no information will be entered.

2 Enter **:w** followed by a space, your new file name, and a RETURN.

```
:w newfilename
```

At the bottom of the screen, the new file name appears in quotes. Next appears the note [New file], followed by the number of lines and the number of characters.

3 Enter the colon command symbol and the **quit** command.

```
:q
```

The system prompt appears at the bottom of the screen. In your directory, the original file exists without change. The new file now appears in your directory; and if any changes had been made, they would be in the new file.

VII Starting a new file without input information.

1 Enter another new file.

```
$ vi newfilename2
```

At the bottom of the screen, the new file name appears in quotes followed by the note [New file]. A line of tildes (~) appears on the left side of the screen. The cursor appears on a line above the tildes. You could enter information at this time.

2 Enter the **write** command.

```
:w
```

The new file name appears in quotes at the bottom of the screen followed by the message "[New file] 1 line, 1 character."

3 Quit the editor.

```
:q
```

The system prompt appears at the bottom of the screen. You now have a new empty file.

VIII Starting a new file with input information.

1 Enter another new file.

```
$ vi newfilename3
```

Wait for a screen display.

2 Write into the new file by typing

```
a
```

then

```
qwerty
```

and press the GO key.

The first letter "a" puts **vi** in the append mode and will not appear on the screen. "qwerty" is the appended information. The GO key is used to end the append mode. You have now entered information into a file.

3 Enter the following **write** and **quit** command:

```
zz
```

The new file name in quotes followed by [New file] 1 line, and 7 characters appears at the bottom of the screen. The system prompt appears on the next line. You now have a new file in your directory containing "qwerty."

Summary

:q	Quit **vi** editor when no changes have occurred since last write.
:q!	Quit **vi** editor and do not save changes since last write.
:w	Write buffer to current file.
:w ***newfile***	Write buffer to *newfile*.
:wq	Write and quit.
ZZ	Write and quit.

Moving Around in a File

Window Movements

Scrolling

The **vi** editor has a number of commands for moving around in the file. The most useful of these is generated by holding down the CODE key and pressing the D key (CODE-**d**). In this chapter, this two-character notation will be used for referring to the CODE key and an associated key. Refer to the discussion of the CODE key in the paragraph covering special purpose keys.

The CODE-**d** command scrolls down in the file. The **d** thus stands for *down*. Many editor commands are mnemonic and this makes them much easier to remember. For instance, the command to scroll up is CODE-**u** with **u** meaning *up*.

If you want to see more of the file below where you are, press CODE-**e** to *expose* one more line at the bottom of the screen. The CODE-**y** command exposes one more line at the top of the screen.

Paging

There are other ways to move around in the file; the keys CODE-**f** and CODE-**b** move *forward* and *backward* a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file using these commands rather than the CODE-**d** and CODE-**u**.

There is a difference between scrolling and paging. If you are trying to read the text in a file, pressing CODE-**f** to move forward a page gives you only a little context at which to look back. Scrolling (CODE-**d**) on the other hand gives more context and functions more smoothly. You can continue to read the text while scrolling is taking place.

Cursor Movements

The editor command set is independent of the terminal you are using. On most terminals with cursor positioning keys (arrow keys), these keys will also work within the editor. If you do not have cursor positioning keys or even if you do, use the **h**, **j**, **k**, and **l** keys as cursor positioning keys. Experienced users of **vi** prefer these keys to arrow keys because they are usually right underneath their fingers.

- **h** moves cursor to the left (like CODE-**h** which is a backspace).
- **j** moves cursor down (in the same column).
- **k** moves cursor up (in the same column).
- **l** moves cursor to the right.

SPACE (the spacebar) moves the cursor right one character and the BACKSPACE key (or CODE-**h**) moves left one character. Also, as noted above, the **l** key will move the cursor to the right. When you type the + key, the cursor advances to the next line in the file at the first nonwhite position on the line. The - key is like + but goes the other way. These are very common keys for moving the cursor up and down lines in the file. The RETURN key has the same effect as the + key.

Note: *If the cursor goes off the bottom (or top) of the screen when using these keys, the text will scroll up (or down) to bring a line at a time into view.*

Moving by Objects

Moving Within a Line

The **w** key will advance the cursor to the first character in the next *word* on the line. The **b** key will move *back* to the first character in a word. The **e** key advances you to the *end* of the current word rather than to the beginning of the next word. If the line has punctuation in it, the **w** and **b** keys stop at each punctuation mark. You can also go backward and forward without stopping at punctuation by using **W** and **B** rather than the lowercase equivalents.

The word movement keys (**e**, **w**, and **b**) wraparound the end of line rather than stopping at the end. You can move to a word on a line below where you are by repeatedly typing **w**.

Low Level Character Motions

The command **fx** finds the next "x" character to the right of the cursor in the current line. A semicolon (;) finds the next instance of the same character. By using the **f** command and then a sequence of ;'s, you can often get to a particular place in a line much faster than with a sequence of word motions or spaces. There is also an **F** command that searches for "x" like **f** but searches backward. The ; command repeats **F** also.

Higher Level Text Objects

In working with a text file, it is often advantageous to work in terms of sentences, paragraphs, and sections.

- The **(** and **)** operations move to the beginning of the previous and next sentence, respectively. A sentence is defined to end at a ., !, or ? that is followed by either the end of a line or two spaces.
- The **{** and **}** operations move to the beginning of the previous and next paragraph, respectively. A paragraph begins after each empty line and also at each of a set of paragraph macros.
- The **[[** and **]]** operations move over sections. Sections in the editor begin after each macro in the sections option (normally, .NH, .SH, .H, and .HU) and each line with a formfeed CODE-l in the first column. Section boundaries are always line and paragraph boundaries.

Moving Around on the Screen

Vi also has commands to take you to the top, middle, and bottom of the screen. The **H** command will take you to the top line (home) on the screen. You can precede it with a number as in **3H**. This will take you to the third line on the screen. Many **vi** commands take preceding numbers and act upon them. **M** takes you to the middle line on the screen, and **L** takes you to the last line on the screen. The **L** command also takes counts, the **5L** command will take you to the fifth line from the bottom.

Moving by Matching Character Strings

Another way to position yourself in the file is by giving the editor a character string to search for. The / character followed by a string of characters terminated by pressing the RETURN key positions the cursor at the next occurrence of this string. Typing an **n** takes you to the next occurrence of this string. Typing an **N** takes you to the next occurrence of this string in the opposite direction. The **?** character will search backward from where you are and is otherwise like /.

These searches will normally wraparound the end of the file and thus find the string even if it is not in the direction you search, provided it is somewhere else in the file. If the search string given the editor is not present in the file, the editor will provide a message at the bottom of the screen that the string was "not found"; and the cursor will return to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with a caret (^). To match only at the end of a line, end the search string with a $. Thus

```
/^first
```

will search for the word "first" at the beginning of a line, and

```
/last$
```

searches for the word "last" at the end of a line.

Miscellaneous Movement Features

The command **G**, preceded by a number, will position the cursor at that line in the file. Thus **1G** will move the cursor to the first line of the file. If you give **G** no count, the cursor moves to the end of the file.

You can also get back to a previous position by using the command `` (two grave accents). This is often more convenient than **G** because it requires no advance preparation. If you accidentally enter **n** or any command that moves you far away from a context of interest, you can quickly get back by typing ``.

Example Two

Note: *This example does not use the RETURN key to end command lines unless specified in the instructions.*

I Window movements.

1 Scrolling.

a Make a copy of a familiar file.

b To access the file copy, enter the **vi** command and the name of your file followed by pressing the RETURN key:

```
$ vi filename
```

The file name will appear in quotes at the bottom of the screen. The first part of the file will then appear on the screen. The cursor will be at the top left character in the file. The file name, number of lines, and number of characters will be displayed on the bottom of the screen.

c Press the CODE key and the D key at the same time.

```
CODE-d
```

The cursor is advanced *down* in the file. (The screen scrolls up adding some new lines at the bottom of the screen.) (For most terminals, eight is a default number of lines; your terminal may scroll more or fewer lines.)

d Press the CODE key and the U key at the same time.

```
CODE-u
```

The screen scrolls down, and the cursor is back at the beginning of the file.

When entering a number with CODE-**d** or CODE-**u**, the number of lines requested last is the permanent amount scrolled when next the CODE-**d** or CODE-**u** is entered without a number.

e Press the CODE key and the E key at the same time.

```
CODE-e
```

One more line is exposed at the bottom of the screen. The cursor remains on the same line.

f Press the CODE key and the Y key at the same time.

```
CODE-y
```

One additional line scrolls on to the top of the screen. The cursor remains on the same line.

2 Paging.

a Press the CODE key and the F key at the same time.

```
CODE-f
```

The screen goes blank momentarily. A new screen image appears containing the last couple of lines of the previous screen image at the top of the screen. These lines are followed by new lines forward in the file. The cursor is at the top of the screen.

b Press the CODE key and the B key at the same time.

```
CODE-b
```

The screen goes blank momentarily. A new screen image appears containing the first couple of lines of the previous screen image at the bottom of the screen. These lines are preceded by new lines backward in the file. The cursor is at the bottom of the screen.

c Practice paging forward and backward in the file. (A number may be entered preceding the input of CODE-**f** or CODE-**b**. The number indicates the number of screen fulls to move forward or backward. It is difficult to keep up with your position in the file when paging more than one screen full.)

II Cursor movements.

1 Cursor positioning keys.

If your terminal has cursor positioning (arrow) keys, perform the following:

a Move the cursor to the right to the end of a line and back with the left and right cursor positioning keys.

b Move the cursor to the bottom of the screen with the down cursor positioning key.

c Add lines to the bottom of the screen with the down cursor positioning key.

d Move the cursor to the top of the screen with the up cursor positioning key.

e Add lines to the top of the screen with the up cursor positioning key.

f Move the cursor to the right 20 character spaces by typing 20 and pressing the right cursor key.

```
20right cursor key
```

If there are less than 20 characters on the line, the cursor will stop on the last character in the line.

g Move the cursor to the left ten character spaces by typing 10 and pressing the left cursor key.

```
10left cursor key
```

If there are less characters to the left of the cursor than ten, the cursor will stop on the first character in the line.

h Move the cursor down ten lines in the file by typing 10 and pressing the down cursor key.

```
10down cursor key
```

The cursor moves down ten lines. The cursor remains ten characters to the left. If the line is less than ten characters long, the cursor is on the last character on the line.

i Move the cursor down 15 more lines.

```
15down cursor key
```

The screen scrolls up and the cursor remains ten characters to the left on the bottom line. If the line is less than ten characters long, the cursor is on the last character on the line.

j Move the cursor up 12 lines by typing a 12 and pressing the up cursor key.

```
12up cursor key
```

The cursor moves up 12 lines.

k Move the cursor up 13 lines.

```
13up cursor key
```

The screen scrolls down and the cursor has returned to its position prior to moving down 25 lines.

2 h, j, k, and **l**

a Move the cursor to the right to the end of a line with lowercase **l**'s and back with lowercase **h**'s. (This performs the same function as the cursor right and left keys.)

b Move the cursor to the bottom of the screen with lowercase **j**'s.

c Add lines to the bottom of the screen with lowercase **j**'s.

d Move the cursor to the top of the screen with lowercase **k**'s.

e Add lines to the top of the screen with lowercase **k**'s.

f Move the cursor to the right 20 character spaces by typing 20 and lowercase **l**.

```
20l
```

If there are less than 20 characters on the line, the cursor will stop on the last character in the line.

g Move the cursor to the left ten character spaces by typing 10 and a lowercase **h**.

```
10h
```

If there are less characters to the left of the cursor than ten, the cursor will stop on the first character in the line.

h Move the cursor down ten lines in the file by typing 10 and a lowercase **j**.

```
10j
```

The cursor moves down ten lines. The cursor remains ten characters to the left. If the line is less than ten characters long, the cursor is on the last character on the line.

i Move the cursor down 15 more lines.

```
15j
```

The screen scrolls up and the cursor remains ten characters to the left on the bottom line. If the line is less than ten characters long, the cursor is on the last character on the line.

j Move the cursor up 12 lines by typing a 12 and a lowercase **k**.

```
12k
```

The cursor moves up 12 lines.

k Move the cursor up 13 lines.

```
13k
```

The screen scrolls down and the cursor has returned to its position prior to moving down 25 lines.

3 BACKSPACE, RETURN, +, -, and SPACEBAR.

a Move the cursor to the right to the end of a line with the SPACEBAR and back with the BACKSPACE key (CODE-**h** will also work).

b Move the cursor to the bottom of the screen with the RETURN key. (The + key will also do this.)

c Add lines to the bottom of the screen with the RETURN key.

d Move the cursor to the top of the screen with the - key.

e Add lines to the top of the screen with the - key.

f Move the cursor to the right 20 character spaces by typing 20 and pressing the SPACEBAR.

```
20
```

If there are less than 20 characters on the line, the cursor will stop on the last character in the line.

g Move the cursor to the left ten character spaces by typing 10 and pressing the BACKSPACE key.

```
10
```

If there are less characters to the left of the cursor than ten, the cursor will stop on the first character in the line.

h Move the cursor down ten lines in the file by typing 10 and pressing the RETURN key.

```
10
```

The cursor moves down ten lines. The cursor moves to the first character space on the line. (The + key will perform the same operation.)

i Move the cursor down 15 more lines.

```
15
```

The screen scrolls up and the cursor moves down 15 lines. The cursor remains in the first character space on the line.

j Move the cursor up 12 lines by typing a 12 and pressing the hyphen (-) key.

```
12-
```

The cursor moves up 12 lines.

k Move the cursor up 13 lines.

```
13-
```

The screen scrolls down. The cursor has returned to its position prior to moving down 25 lines except the cursor has moved to the first character space on the line.

III Moving by objects.

1 Moving by characters.

a Place the cursor at the beginning of a long line containing a large number of entries of the same character. (The cursor does not have to be on the first character.)

b Type a lowercase **f** and **a**.

```
fa
```

The cursor moves to the right to the first "a" on the line.

c Type a semicolon.

```
;
```

The cursor moves to the right to the next "a" on the line.

d Continue to type semicolons until the last a on the line is reached.

After reaching the last a on the line, typing more semicolons will not move the cursor any further.

e Type a comma.

```
,
```

The cursor moves back to the next to last a on the line.

f Move the cursor to the end of the line. (The cursor does not have to be on the last character on the line.)

g Type an uppercase **F** and a lowercase character that appears several times on the line, for example an e.

```
Fe
```

The cursor moves to the left to the next "e" on the line.

h Type a semicolon.

```
;
```

The cursor moves to the left to the next "e" on the line.

i Continue to type semicolons until the first "e" on the line is reached.

After reaching the first "e" on the line, typing more semicolons will not move the cursor any further on the line.

j Type a comma.

```
,
```

The cursor moves to the right to the second "e" on the line.

k Move the cursor to the beginning of the line.

l Type a 3 followed by a lowercase **f** and **a**. (This step assumes there are at least three a's on your test line.)

```
3fa
```

The cursor moves to the right to the third a on the line.

m Type a 2 followed by a semicolon. (This step assumes there are at least five "a"'s on your test line.)

```
2;
```

The cursor moves to the right two more "a"'s.

n Type a 4 followed by a comma.

```
4,
```

The cursor moves to the left four "a"'s.

o Move backward on the line searching for a character using numbers, **F**'s, semicolons, and commas.

The cursor moves as in the previous three steps except in the opposite direction.

p Practice moving the cursor to desired characters by using **f**'s, **F**'s, numbers, semicolons, and commas.

q Perform the steps just performed for moving to characters except subtitute a **t** for the **f** and a **T** for the **F**.

When typing the **t**, the cursor in each step will move to the character to the left of the character being searched for. When typing the **T**, the cursor in each step will move to the character to the right of the character being searched for.

2 Moving by words.

a Type an uppercase **W**.

```
W
```

The cursor moves to the right to the first character in the next word on the line. If the cursor is on any character in the last word in the line, the cursor will move to the first character in the first word in the next line down.

b Type a lowercase **w**.

```
w
```

The cursor moves as before except the cursor stops at punctuation and other non-alphabetical and non-numerical characters.

c Type an uppercase **B**.

```
B
```

The cursor moves to the left to the first character in the next word in the line. If the cursor is on any character in the first word in the line, the cursor will move to the first character in the last word in the next line up.

d Type a lowercase **b**.

```
b
```

The cursor moves as before except the cursor stops at punctuation and other non-alphabetical and non-numerical characters.

e Type an uppercase **E**.

```
E
```

The cursor moves to the right to the last character in the current word. If the cursor is on the last character in a word, the cursor will move to the last character in the next word to the right. If the cursor is on the last character of the last word in a line, the cursor will move to the last character of the first word on the next line down.

f Type a lowercase **e**.

```
e
```

The cursor moves as before except the cursor stops at punctuation and other non-alphabetical and non-numerical characters.

g Type a 5 and an uppercase **W**.

```
5W
```

The cursor moves to the right to the first character of the fifth word. The cursor will move down to the next line(s) if five remaining words are not in the current line.

3 Moving by sentences. The following feature of the **vi** editor will work only if two spaces are used between the end of sentence punctuation and the beginning of the next sentence.

a Place the cursor at the beginning of a paragraph.

b Enter a right parenthesis.

```
)
```

The cursor moves down to the first character of the next sentence.

c Enter a left parenthesis.

```
(
```

The cursor moves up to the first character of the previous sentence.

d Type a 4 and a right parenthesis.

```
4 )
```

The cursor moves down in your file to the first character of the fifth sentence.

e Type a 3 and a left parenthesis.

```
3 (
```

The cursor moves up in your file to the first character of the second sentence in the paragraph.

f Move the cursor to a sentence in the next paragraph by typing single right parenthesis in succession.

```
)
```

The cursor stops at the beginning of an empty space. The space between paragraphs counts as one sentence.

g Practice moving the cursor by typing numbers, right parentheses, and left parentheses.

4 Moving by paragraphs.

a Move the cursor to the beginning of your file by using any of the following:

```
CODE-u
CODE-b
-
up cursor key
k
```

b Type a right brace.

```
}
```

The cursor moves down to the beginning of the next empty line.

c Type a left brace.

```
{
```

The cursor moves up to the beginning of the first line in your file.

d Type a 3 and a right brace.

```
3}
```

The cursor moves down to the third empty line space in your file.

e Type a 2 and a left brace.

```
2{
```

The cursor moves up to the first empty line space in your file.

f Practice moving by paragraphs using numbers, left braces, and right braces.

5 Moving to the beginning and ending of lines.

a Place the cursor in the middle of a line.

b Type a zero.

```
0
```

The cursor moves to the first character space on the current line.

c Type a dollar sign.

```
$
```

The cursor moves to the last character in the current line.

d Move the cursor back to the middle of the line.

e Type a 2 and a dollar sign.

```
2$
```

The cursor moves to the last character of the next line. When using numbers and $'s, the line count includes the current line.

IV Moving by matching character strings.

1 Move the cursor to the first line in your file.

2 Select a word to "search for" that appears on the screen.

3 Type a forward slash.

```
/
```

The slash appears at the bottom left of the screen.

4 Type your *word* and press the RETURN key.

```
word
```

Your *word* follows the slash at the bottom of the screen. After pressing the RETURN key, the cursor moves to the first character of the first appearance of your word on the screen.

5 Find the next occurrence of your word by typing an **n**.

```
n
```

A forward slash appears at the bottom of the screen. The cursor moves forward in your file to the first character in the next appearance of your word.

6 Move the cursor to the bottom line of the screen.

7 Select another word to search for that appears on the screen.

8 Perform a backward search by typing the following:

```
?word2
```

Your second word follows the question mark at the bottom of the screen. After pressing the RETURN key, the cursor moves to the first character of the first appearance of your word in a backward direction in your file.

9 Find the next occurrence of your word by typing an **n**.

```
n
```

A backward slash appears at the bottom of the screen. The cursor moves backward in your file to the first character in the next appearance of your word.

10 Type an uppercase **N**.

```
N
```

A forward slash appears at the bottom of the screen. The cursor moves forward in your file to the first character of your *word2* (back to the same word in the previous step). (A forward slash and pressing the RETURN key will also reverse the direction of the search.)

11 Select a word in your file that is the first word on a line.

12 Move the cursor to a position in your file ahead of the word.

13 Type a forward slash, a caret, and your third word, followed by pressing the RETURN key.

```
/^word3
```

The information typed appears at the bottom of the screen. After pressing the RETURN key, the cursor moves forward to the next appearance of your word at the beginning of the line.

14 Select a word in your file that is the last word on a line.

15 Move the cursor to a position in your file ahead of the word.

16 Type a forward slash, your fourth word, and a dollar sign followed by pressing the RETURN key.

```
/word4$
```

16 The information typed appears at the bottom of the screen. After pressing the RETURN key, the cursor moves forward to the next appearance of your word at the end of a line.

17 Type a forward slash followed by a word that you know is not in your file. Press the RETURN key.

```
/word5
```

At the bottom of the screen the following message appears:

```
/Pattern not found
```

The word initially searched for does not have to appear on the screen.

String searches normally search to the end or beginning of the file and continue searching from the opposite end of your file until your word is found.

Summary

BACKSPACE	Moves cursor one position to the left.
SPACEBAR	Moves cursor one position to the right.
CODE-b	Moves backward to previous page.
CODE-d	Scrolls down in the file.
CODE-e	Exposes another line at bottom of screen.
CODE-f	Moves forward to next page.
CODE-g	Determines state of file. (file name, current line number, number of lines in the buffer, and percent of way through the buffer)
CODE-h	Moves cursor one space to the left.
CODE-u	Scrolls up in the file.
CODE-y	Exposes another line at the top of screen.
+	Advances cursor to beginning of next line.
-	Moves cursor to beginning of previous line.
/	Searches forward for a character string.
?	Searches backward for a character string.
B	Moves cursor backward a word ignoring punctuation.
G	Moves cursor to specified line or to last line if default.
H	Moves cursor to top line (home) on screen.
M	Moves cursor to middle line on screen.
N	Repeats scan for next instance of / or ? pattern, but in the reverse direction of the last search.
L	Moves cursor to last line on screen.
W	Moves cursor forward a word ignoring punctuation.
b	Moves cursor backward a word.
e	Moves cursor to end of current word.
h	Moves cursor to the left.
j	Moves cursor down (in same column).
k	Moves cursor up (in same column).

l	Moves cursor to the right.
n	Repeats scan for next instance of / or ? pattern.
w	Moves cursor forward a word.

Creating New Text

Append or Insert Mode

One of the most useful commands is the **insert** (**i**) command. After typing **i**, everything typed until you press the GO key is inserted into the file. If you are on a dumb terminal, it will seem that some of the characters in your line have been overwritten; but they will reappear when you press the GO key. A similar command is the **append** (**a**) command.

- **i** places text to the left of the cursor
- **a** places text to the right of the cursor.

When working with the text of a single line, a caret (^) moves the cursor to the first nonwhite position on the line, and a **$** moves it to the end of the line. Thus, **$a** will append new text at the end of the current line. **A** is equivalent to **$a**. ^**i** will insert new text at the beginning of the current line. **I** is equivalent to ^**i**.

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lowercase key and the other is given by an uppercase key. The uppercase key often differs from the lowercase key in its sense of direction (the uppercase key moves backward and/or up, and the lowercase key moves forward and/or down).

It is often the case that you want to add new lines to the file you are editing before or after some specific line in the file. The **o** command creates a new line after the line you are on, and the **O** command creates a new line before the line you are on. When you create a new line in this way, the text you type is inserted on the new line until you press the GO key.

Whenever you are typing in text, you can give many lines of input or just a few characters. To type in more than one line of text, press the RETURN key at the middle of your input. A new line will be created for text, and you can continue to type. If you are on a slow and dumb terminal, the editor may choose to wait to redraw the tail of the screen and will let you type over the existing screen lines. This avoids the lengthy delay that would occur if the editor attempted to keep the tail of the screen always up to date. The tail of the screen will be fixed and the missing lines will reappear when you press the GO key.

While inserting new text, use the characters normally used at the system command level (usually CODE-**h** or #) to backspace
over the last character typed; and the character used to kill input lines (usually @, CODE-**x**, or CODE-**u**) can be used to erase the input typed on the current line. In fact, CODE-**h** (backspace) always erases the last input character regardless of what your erase character is. CODE-**w** will erase a whole word and leave you after the space following the previous word. It is useful for quickly backing up in an insert. The following conditions should be noted:

- When you backspace during an insertion, the characters you backspace over are not erased. The cursor moves backward, and the characters remain on the display. This is often useful if you are planning to type in something similar. In any case, the characters disappear when you press the GO key. If you want to delete them immediately, press the GO key and then **a** again.
- You cannot erase characters that you did not insert, and you cannot backspace around the end of a line. If you need to back up to the previous line to make a correction, just press the GO key and move the cursor back to the previous line. After making the correction, return to where you were and use the **insert** or **append** command again.

Example Three

1 Start a new file as follows:

```
$ vi newfilename
```

At the bottom of the screen, the new file name appears in quotes followed by [*New file*]. A line of tildes (~) appear on the left side of the screen. The cursor appears on a line above the tildes.

2 Type a lowercase **a**.

```
a
```

Nothing happens. The editor is in the append mode.

3 Type the following paragraph (including errors). Do not worry about making typing mistakes. If your mistakes are too extensive, type :**q**! and start over. The backspace can be used to type over your errors. Type two spaces between sentences when the end of the sentence is not at the end of a line:

```
When you have successfully logged in, a program
called the shell is listening to your terminal.
The shell read the line you type, splits it into
a command name and its arguments, and the
command.  A simply an executable program.
located in a file.  Normally, the shell looks first
in your current directory for a program with your
directories.  There is nothing special about
system-provided commands except that they are kept
for the shell to find them there.
```

4 Press the GO key.

The cursor moves to the left one space. The editor is out of the append mode. The previous steps could have been performed by typing a lowercase i, *text*, and pressing the GO key.

5 Move the cursor to the third line from the top of the file.

6 Move your cursor to the d in the word "read."

7 Type a lowercase **a**.

```
a
```

The cursor moves one space to the right.

8 Type a lowercase **s**.

```
s
```

The s is added following "read." The remainder of the sentence is moved to the right one space.

9 Press the GO key.

The cursor moves to the left one space. The editor is out of the append mode.

10 Move the cursor to the fourth line from the top of the file.

11 Move your cursor to the space following the second "and."

12 Type a lowercase **a**.

```
a
```

The cursor moves one space to the right.

13 Type the following word and then type a space:

```
executes
```

The typed information is added following "and." The "the" is moved to the right as you type.

14 Press the GO key.

The cursor moves to the left one space. The editor is out of the append mode.

15 Move the cursor to the fifth line from the top of the file.

16 Move the cursor to the first m in "command."

17 Type a lowercase i.

```
i
```

The cursor remains in the same character space.

18 Type a lowercase **o**.

```
o
```

The o is inserted before the first m. The remainder of the sentence moves over one character space.

19 Press the GO key.

The cursor moves to the left one space. The editor is out of the insert mode.

20 Move the cursor to the s in simply.

21 Type a lowercase **i**.

```
i
```

The cursor remains in the same character space.

22 Type the following text and then type a space:

```
command is
```

The typed information is inserted before "simply." The remainder of the sentence moves over as you type.

23 Press the GO key.

The cursor moves to the left one space. The editor is out of the insert mode.

24 Move the cursor to any character in the seventh line from the top of the file.

25 Type a lowercase **o**.

```
o
```

A blank line is added following the line you are on. The cursor appears at the first character space in the empty line.

26 Type the following text:

```
command name, and if none is there, then in system
```

27 Press the GO key.

The cursor moves to the left one space. The editor is out of the open mode.

28 Move the cursor to any character in the last line in your file.

29 Type an uppercase **O**.

```
O
```

A blank line is added preceding the line your cursor is on. The cursor appears at the first character space in the empty line.

30 Type the following text:

```
in directories where the shell can find them.
You can (press the RETURN key)
also keep commands in your own
directories and arrange
```

31 Press the GO key.

The cursor moves to the left one space. The editor is out of the open mode.

32 Correct your typing errors (if any) where possible, using **a**, **i**, **o**, and **O**.

33 The file you have just completed should appear as follows:

```
When you have successfully logged in, a program
called the shell is listening to your terminal.
The shell reads the line you type, splits it into
a command name and its arguments, and executes the
command.  A command is simply an executable program
located in a file.  Normally, the shell looks first
in your current directory for a program with your
command name, and if none is there, then in system
directories.  There is nothing special about
system-provided commands except that they are
kept in directories where the shell can find them.
You can also keep commands in your own directories
and arrange for the shell to find them there.
```

34 Save this file for use in other examples by typing the following:

```
ZZ
```

At the bottom of the screen, your file name appears in quotes followed by [*New file*], the number of lines, and the number of characters.

Summary

^w	Erases a word during an insert.
erase	Erases a character during an insert (usually **^h** or #).
kill	Kills the insert on this line (usually @, **^x**, or **^u**).
O	Opens and inputs new line(s) above the current line.
a	Appends text after the cursor.
A	Appends text after the line on the same line.
i	Inserts text before the cursor.
I	Inserts text before the line on the same line.
o	Opens and inputs new line(s) below the current line.

Deleting Text

Deleting and Recovering With Undo

Suppose the last change you made was incorrect; you can use the insert, delete, and append commands to put the correct material back. However, since it is often the case that we regret a change or make a change incorrectly, the editor provides an **undo** (**u**) command to reverse the last change made. A **u** also undoes a previous **u**.

The **u** command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the original state of the line back. The **U** command restores the current line to the state before you started changing it.

You can recover text that you deleted even if the **u** command will not bring it back; see the subsection on recovering lost text in this chapter.

Deleting Characters, Words, Sentences, Paragraphs, and Lines

You can make small corrections in existing text quite easily. Use the arrow keys to find a character or get near a character with the word motion keys and then either backspace (press the BACKSPACE key, CODE-**h**, or just **h**) or SPACE (using the space bar) until the cursor is on the wrong character. If the character is not needed, then type the **x** key; this deletes the character from the file. It is analogous to the way you x out characters when you make mistakes on a typewriter. Counts are also useful with **x** to specify the number of characters to be deleted.

When operating on the text in a line, it is often desirable to delete characters including the first instance of a character. Try **df***x* for some *x* and notice that the *x* character is deleted. Undo this with **u** and then try **dt***x* (the **t** stands for "to") to delete up to the next *x*, but not the *x*. The command **T** is the reverse of **t**.

Making Corrections with Operators

The **d** key acts as a delete operator.

- The **dw** command deletes a following word.
- The **db** command deletes a preceding word.
- The **d)** command will delete the rest of the current sentence. The **d(** command will

  delete the previous sentence if you are at the beginning of the current sentence

  or

  delete the current sentence up to your present position if you are not at the beginning of the current sentence.

- The **d}** command will delete the rest of the current paragraph. The **d{** command will

  delete the previous paragraph if you are at the beginning of the current paragraph

  or

  delete the current paragraph up to your present position if you are not at the beginning of the current paragraph.

An uppercase **D** will delete the current line and leave the cursor on a blank line.

Operating on Lines

It is often the case that you want to operate on lines. A line can be deleted by typing **dd**, the **d** operator twice. This will delete the line. If you are on a dumb terminal, the editor may erase the line on the screen replacing it with a line with only an @ on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen which would be necessary to close up the hole created by the deletion on a terminal without a delete line capability.

You can delete more than one line by preceding the **dd** with a count (**5dd** deletes five lines). You can also give a command like **dL** to delete all the lines up to and including the last line on the screen or **d3L** to delete through the third from the bottom line. The / (search) after a **d** will delete characters from the current position to the point of the match.

Note: *Vi lets you know when you change a large number of lines so that you can see the extent of the change. It will also always tell you when a change you make affects text that you cannot see.*

Example Four

1 Access the file created for Example Three.

```
$ vi filename
```

If this file was not saved, enter the file as shown at the end of Example Three.

2 Add the following text to your file (including errors). Add a blank line at the beginning of the text. Type two spaces between sentences when the end of the sentence is not at the end of a line:

```
(blank line)
The command name is the firrst word on an input liuone
to shell; the commanding and arguments are separated
from one another the by space and/or tab characters.
The quick brown fox jumped over the lazy
dog's back.

When a program terminates, the shell will ordinarily
regain control and type a $ at you to indicate that
Merry Christmas
Happy New Year
Aint apple pie wonderful.
itcan be is ready for another command.
The sheldelete charactersl has many
other capabilities described the big apple
in detail in the user's manual under sh. Carry me back
to ole Virginny.
```

Your file now contains three paragraphs.

3 Move the cursor to the second word in the second paragraph (commabnd).

4 Move the cursor to the b.

5 Remove the b by typing a lowercase **x**.

```
x
```

The b is deleted. The remainder of the line moves one character space to the left.

6 Undo the deletion of the b by typing a lowercase **u**.

```
u
```

The b returns to its former position in the word command.

7 Undo the undo by typing another **u**.

```
u
```

The b is removed for the second time.

8 Move the cursor to the s in firrst.

9 Delete the second r by typing the following:

```
x
```

The r is deleted. The remainder of the line moves one character space to the left.

10 Move the cursor to the n in liuone.

11 Delete the uo by typing the following:

```
2x
```

The uo is deleted. The remainder of the line moves two character spaces to the left.

12 Undo all of the deletions on the line by typing an uppercase **U**.

```
U
```

The line returns to its original state.

13 Undo the line undo by typing a lowercase **u**.

```
u
```

The corrected line is restored.

14 Move the cursor to the next line. Move the cursor to the i in the word commanding.

15 Remove the ing by typing **3x**.

```
3x
```

The ing is deleted. The remainder of the line moves three character spaces to the left.

16 Move the cursor to the "the" in the third line in the second paragraph.

17 Remove the "the" by entering the **delete word** command.

```
dw
```

"the" is removed. The extra space is also removed. The remaining characters move to the left.

18 Experiment with the undo as you perform the remainder of this procedure.

19 Move the cursor to the q in the fourth line in the second paragraph.

20 Delete the two words "quick brown" by typing the following:

```
d2w
```

"quick brown" is deleted. The remaining characters move to the left.

21 Move the cursor to any character in the fourth line in the second paragraph.

22 Delete the line the cursor is on by typing the following:

```
dd
```

The line is deleted. The following lines move up one line.

23 Move the cursor to any character in the third line in the third paragraph.

24 Delete the line the cursor is on and the next line by typing the following:

```
2dd
```

The two lines are deleted. The following lines move up two lines.

25 Delete the new third line by typing the following:

```
D
```

The third line is deleted. A blank line remains.

26 Move the cursor to the space before the "the" in the fifth line of the third paragraph.

27 Remove the remainder of the line by typing the following:

```
d$
```

"the big apple" is deleted.

28 Move the cursor to the space before "Carry" in the next line.

29 Delete the remainder of the line and the next line by typing the following:

```
d2$
```

Carry me back to ole Virginny is deleted.

30 Move the cursor to the c in can in the third line of the third paragraph.

31 Remove "can be" by typing the following:

```
dfe
```

All characters from the cursor through the first e are deleted.

32 Move the cursor to the d in "delete characters" in the fourth line of the third paragraph.

33 Delete "delete characters" by typing the following:

```
d2tl
```

"delete characters" is deleted.

34 Move the cursor to the u in you in the first line of the first paragraph.

35 Delete the character the cursor is on plus the next nine characters by typing the following and pressing the SPACEBAR:

```
d10
```

Ten characters including spaces are deleted through the first c in successfully. Undo this deletion.

This can be done in the reverse direction with the BACKSPACE.

36 Experiment with deleting characters using **df**, **dt**, **d**-SPACEBAR, and **d**-BACKSPACE using numbers following the **d**.

37 Move the cursor to any character in the first line of the first paragraph.

38 Delete the line the cursor is on through the fifth line in the file by typing the following:

```
d5G
```

The first five lines are deleted. The remaining lines move up to replace these lines. Undo this deletion.

39 Move the cursor to any character in the eighth line in the first paragragh.

40 Delete this line plus all lines to the end of the file by typing the following:

```
dG
```

The lines are deleted. Tildes follow the line the cursor is on. The first seven lines remain in the file. Undo this deletion.

41 Move the cursor to any character in the second line of the first paragraph.

42 Delete this line plus the next two lines by typing the following command and pressing the RETURN key:

```
d2
```

The lines are deleted. Undo this deletion. Typing a **d2+** will delete three lines in the same manner.

43 Move the cursor to any character in the fourth line of the first paragraph.

44 Delete the line the cursor is on plus all lines through the last line displayed on the screen by typing the following:

```
dL
```

The lines are deleted. The remaining lines in the file move up. Undo this deletion.

45 Move the cursor to any character in the second line of the third paragraph.

46 Delete the line the cursor is on plus all lines through the middle line displayed on the screen by typing the following:

```
dM
```

The lines are deleted. The remaining lines in the file fill in the space occupied by the deleted lines. Undo this deletion.

47 Move the cursor to the fourth line from the top of the screen.

48 Delete the line the cursor is on through the line at the top of the screen by typing the following:

```
dH
```

The four lines are deleted. The remaining lines in the file close up. Undo this deletion.

49 Experiment with deleting lines using **dH**, **dM**, **dL**, and **dG** using numbers following the **d**. Undo all your deletions.

50 Move the cursor to the "A" in the third sentence in the first paragraph.

51 Temporarily remove this sentence by typing the following:

```
d)
```

The sentence is deleted. The remainder of the next line is moved up to fill in the line the cursor is on. The remaining lines move up one line without change. Undo this deletion.

52 Move the cursor to the N in the next line.

53 Temporarily remove the previous sentence by typing the following:

```
d(
```

The previous sentence is deleted. The remainder of the file moves up as in last deletion. Undo this deletion.

54 Move the cursor to the A in the third sentence.

55 Temporarily delete the next three sentences by typing the following:

```
d3)
```

The next three sentences are deleted. "You can" fills in the line the cursor is on. The remainder of the file moves up without change. Undo this deletion.

You can delete more than one previous sentence in the same manner using **d(** with the number of sentences before or after the **d**.

56 Move the cursor to the s in the word simply in the third sentence.

57 Temporarily delete the remainder of this sentence by typing the following:

```
d)
```

Beginning with the word simply the remainder of the sentence is deleted. The remainder of the next line fills in the line the cursor is on. The remaining portion of the file moves up without change. Undo this deletion.

You can remove more than one sentence in this manner in both the forward and reverse directions.

58 Move the cursor to the blank line before the first line of the second paragraph.

59 Temporarily remove this paragraph by typing the following:

```
d}
```

The second paragraph is deleted. The last paragraph is moved up with a blank line between the two remaining paragraphs. Undo this deletion.

60 With the cursor on the same blank line, remove the previous paragraph by typing the following:

```
d{
```

The first paragraph is deleted. The cursor is on a blank line that is now the first line in the file. Undo this deletion.

61 With the cursor on the first line in the file, delete the first two paragraphs by typing the following:

```
d2}
```

One paragraph remains in the file. The cursor is on a blank line at the beginning of the file. Undo this deletion.

62 Move the cursor to the space before the third paragraph.

63 Delete the first two paragraphs by typing the following:

```
d2{
```

One paragraph remains in the file. The cursor is on a blank line at the beginning of the file. Undo this deletion.

64 Move the cursor to the "A" in the third sentence in the first paragraph.

65 Remove the remainder of the first paragraph by typing the following:

```
d}
```

The remainder of the paragraph is deleted. Paragraphs two and three move up with no change. A blank line remains between paragraphs one and two. Undo this deletion.

You can delete parts of paragraphs in both the forward and reverse directions. You can also delete more than one paragraph using this method.

66 Move the cursor to the s in successfully in the first line of the first paragraph.

67 Delete all information from the cursor location up to the "A" in the fifth line by typing the following command and pressing the RETURN key:

```
d/A
```

All the information from the space after "have" up to "A" is deleted. Beginning with "A", the remainder of the fifth line fills in the first line. The remainder of the file moves up with no change. Undo this deletion.

68 Experiment with the various methods of deleting and the undo command.

69 Save this file for use in the next example. The file should appear as follows:

```
When you have successfully logged in, a program called
the shell is listening to your terminal.  The shell
reads the line you type, splits it into a command name
and its arguments, and executes the command.  A command
is simply an executable program located in a file.
Normally, the shell looks first in your current
directory for a program with your command name, and if
none is there, then in system directories.  There is
nothing special about system-provided commands except
that they are kept in directories where the shell can
find them.  You can also keep commands in your own
directories and arrange for the shell to find them
there.

The command name is the first word on an input line to
shell; the command and its arguments are separated from
one another by space and/or tab characters.

When a program terminates, the shell will ordinarily
regain control and type a $ at you to indicate that it
is ready for another command.  The shell has many other
capabilities described in detail in the user's manual
under sh.
```

Summary

D	Deletes the current line.
U	Undoes the deletes made to the current line.
d	Deletes the specified object (word, space, and so on).
dd	Deletes a line.
u	Undoes the last delete.
x	Deletes a character.

Changing Text

Change (Delete and Insert)

The **c** key acts as a change operator. As with the delete operator, you can change any object that the visual editor recognizes (characters, words, lines, sentences, paragraphs). For example, the **cw** command changes the text of a single word.

When you type in a **change** command, the end of the text to be changed is marked with the $ character to indicate that a change is now anticipated up to the $ character. You are now placed in the insert mode so that anything you type is entered into the buffer. You terminate the insert mode by pressing the GO key.

To summarize, **change** commands in **vi** act to delete text objects and then place you in the insert mode. As with other inserts, you get back to the command mode by pressing the GO key.

Note: *Vi lets you know when you change a large number of lines so that you can see the extent of the change. It will also always tell you when a change you make affects text which you cannot see.*

Changing Characters, Words, Sentences, Paragraphs, and Lines

The simplest change you can make is to change one character. To do this **vi** has two possible commands, **r** and **s**. If a character is incorrect, you can replace it with the correct character by giving the **r***x* command, where *x* is the correct character. If the incorrect character should be replaced by more than one character, make the following command entry and press the GO key:

```
sstring
```

which substitutes a *string* of characters and ends with GO. If there are a small number of characters wrong, precede **s** with a count of the number of characters to be replaced.

- The **cw** command allows you to change a following word.
- The **cb** command allows you to change a preceding word.
- The **c)** command allows you to change the rest of the current sentence. The **c(** command allows you to

  change the previous sentence if you are at the beginning of the current sentence

  or

  change the current sentence up to your present position if you are not at the beginning of the current sentence.
- The **c}** command allows you to change the rest of the current paragraph. The **c{** command allows you to

  change the previous paragraph if you are at the beginning of the current paragraph

  or

  change the current paragraph up to your present position if you are not at the beginning of the current paragraph.

An uppercase **C** allows you to change information from the cursor to the end of the current line. The line will be displayed while you overwrite. Repeating the **c** operator twice (**cc**) will change a whole line erasing its previous contents and replacing them with text you type before pressing the GO key. The **S** command is a convenient synonym for **cc**.

You can change more than one character, word, line, sentence, or paragragh by preceding the change entry with a number. For example, preceding **cc** with a count of five (**5cc**) allows you to change five lines.

You can also give a command like **cL** to change all the lines up to and including the last line on the screen or **c3L** to change through the third line from the bottom line. Using the / (search) after a **c**, allows you to change characters from the current position to the point of the match.

Example Five

1 Access the file created for Example Four.

```
$ vi filename
```

If this file was not saved, enter the file as shown at the end of Example Four.

2 Add the following text to your file (including errors). Add a blank line at the beginning of the text. Type two spaces between sentences when the end of the sentence is not at the end of a line:

```
(blank line)
The fole system is arriged in a Hierarchy of
four score and seven years ago administrator gave you
a user wrong, he or she also created a directory
for you (ordinarily for a same name as your user
This text is not the correct text.
This line is to be changed
along with this line.
assumed to be in that directory.  Incorrect information
owner of this directory, now is the time for all
good men to come to the aid of their country.
Permotion for you your will with other directories
and filechange charactersen granted or denied to you by
their respective owners or by the system administrator.
```

There are now four paragraphs in your file.

3 Move the cursor to the f in fole in the first line of the fourth paragraph.

4 Move the cursor to the o.

5 Change the o to an i by typing a lowercase **r** followed by a lowercase i.

```
ri
```

The o is changed to an i.

6 Undo the change by typing a lowercase **u**.

```
u
```

The o returns in the word file.

7 Undo the undo by typing another **u**.

```
u
```

The i is returned for the second time.

8 Move the cursor to the i in "arriged" in the same line.

9 Substitute an for i by entering the **substitute** command, "an," and pressing the GO key:

```
san
```

The new word is "arranged." The remainder of the line moves one character space to the right.

10 Move the cursor to the H in Hierarchy in the first line.

11 Change the H to h by typing the following:

```
~
```

The H is changed to h.

12 Undo all of the changes on the line by typing an uppercase **U**.

```
U
```

The line returns to its original state.

13 Undo the the line undo by typing a lowercase **u**.

```
u
```

The corrected line is restored.

An uppercase **U** will now not perform a line undo.

14 Move the cursor to the first character in next line.

15 Replace "four score and seven years ago" by entering the **replace** command, the replacement text, and pressing the GO key:

```
Rdirectories.  When the system
```

The new text is typed over the old text. Remove the remaining o by placing the cursor on the o and typing an **x**. The new text returns.

16 Move the cursor to the w in "wrong" in the third line in the fourth paragraph.

17 Change the word "wrong" to "name" by entering the **change word** command, "name," and pressing the GO key:

```
cwname
```

A $ sign is placed where the g was at the end of the word when cw is typed. "name" is typed over "wron." When the GO key is pressed, the $ disappears. The remaining characters move to the left one space.

18 Experiment with the "undo" and "undoing the undo" as you perform the remainder of this procedure.

19 Move the cursor to the f in for in the fourth line in the fourth paragraph.

20 Change the two words "for a" to "with the" by typing the following and then pressing the GO key:

```
c2wwith the
```

A $ sign is placed where the "a" was when **cw** is typed. "with the" is typed over "for $." When the escape key is pressed, the remaining characters move to the right.

Text not affected by a **change** command fills in or expands as when GO is pressed.

21 Move the cursor to any character in the fifth line in the fourth paragraph.

22 Change the line the cursor is on by typing the following and pressing the GO key:

```
ccname, and known as your login or home directory).
```

When **cc** is typed, the line is deleted leaving the cursor at the beginning of a blank line. The new text is typed into the blank line.

23 Move the cursor to any character in the sixth line in the fourth paragraph.

24 Change the line the cursor is on and the next line by typing the following and pressing the GO key:

```
2ccWhen you log in, that directory becomes your current
(Press the RETURN key)
directory, and any file name you type is by default
```

The two lines are deleted following the entry of the second **c**. The following lines move up one line leaving one blank line with the cursor at the beginning.

An unlimited number of lines can replace the original two lines.

25 Move the cursor to the I in Incorrect in the eighth line of the fourth paragraph.

26 Change the remainder of the line by entering the **change** command, typing the new text, and pressing the GO key:

```
CBecause you are the
```

As soon as the **C** is typed, a $ replaces the last character in the sentence. The new text is typed over the old text.

27 Move the cursor to the n in now in the next line.

28 Change the remainder of the line and the next line by entering the **change** command, typing the new text, and pressing the GO key:

```
C2you have full permissions
(Press the RETURN key)
to read, write, alter, or destroy its contents.
```

As soon as the **C** is typed, the remainder of the line the cursor is on and the next line disappear. The cursor remains at its original position until the new text is typed in.

29 Move the cursor to the o in Permotion in the eleventh line of the fourth paragraph.

30 Change "otion for you" by typing the following and pressing the GO key:

```
cfuissions to have
```

As soon as **cfu** is typed, a $ replaces the u in you. The new text is typed over the old text. The remainder of the line moves right.

31 Move the cursor to the c in change characters in the twelth line of the fourth paragraph.

32 Change "change characters" by typing the following and pressing the GO key:

```
c3tes will have be
```

A $ replaces the s in characters as soon as **c3fe** is typed. The new text is typed over the old text. The remainder of the line moves left when GO is typed.

33 Move the cursor to the u in you in the first line of the first paragraph.

34 Change the character the cursor is on plus the next nine characters by typing the following and pressing the GO key:

```
c10(Press the SPACEBAR)
xxType any number of charactersxx
```

10 characters including spaces are changed through the first c in successfully.

This can be done in the reverse direction with the backspace. Undo this change.

35 Experiment with changing characters using **cf**, **ct**, **c**-SPACEBAR, and **c**-BACKSPACE using numbers following the **c**.

36 Move the cursor to any character in the first line of the first paragraph.

37 Change the line the cursor is on through the fifth line in the file by typing the following:

```
c5GxxxxxType any number of linesxxxxxxxxxxxx
(press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxx
```

The first five lines are deleted. The remaining lines move up to replace these lines. The cursor is on a blank line. Undo this change.

38 Move the cursor to any character in the eighth line in the first paragragh.

39 Change this line plus all lines to the end of the file by typing the following and pressing the GO key:

```
cGxxxxxxType any number of linesxxxxxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

After typing **cG**, the desired lines are deleted. The cursor is on a blank line at the end of the file. Tildes follow the line the cursor is on. The first seven lines remain in the file. The new text is added to the end of the file. Undo this change.

40 Move the cursor to any character in the second line of the first paragraph.

41 Change this line plus the next two lines by typing the following and pressing the GO key:

```
c2
(Press the RETURN key)
xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

After typing **c2** and pressing the RETURN key, the lines are deleted. The cursor is on a blank line following the first line in the file. The new text is added following the first line. Undo this change.

Typing a **c2**+ will change three lines in the same manner.

42 Move the cursor to any character in the fourth line of the first paragraph.

43 Change the line the cursor is on plus all lines through the last line displayed on the screen by typing the following and pressing the GO key:

```
cLxxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

After typing **cL**, the desired lines are deleted. The cursor is on a blank line. The remaining lines in the file move up. The new text is added following the third line. Undo this change.

44 Move the cursor to any character in the second line of the third paragraph.

45 Change the line the cursor is on plus all lines through the middle line displayed on the screen by typing the following and pressing the GO key:

```
cMxxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

After typing **cM**, the desired lines are deleted. The cursor is on a blank line. The remaining lines in the file move up. The new text is added following the first line. Undo this change.

46 Move the cursor to the fourth line from the top of the screen.

47 Delete the line the cursor is on through the line at the top of the screen by typing the following and pressing the GO key:

```
cHxxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

After typing **cH**, the four lines are deleted. The cursor is on a blank line. The remaining lines in the file move up. The new text is added preceding the new first line. Undo this change.

48 Experiment with deleting lines using **cH**, **cM**, **cL**, and **cG** using numbers following the **c**. Undo all your changes.

49 Move the cursor to the "A" in the third sentence in the first paragraph.

50 Temporarily change this sentence by typing the following and pressing the GO key:

```
c)xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

After typing **c)**, the sentence is deleted. The remainder of the next line is moved up to fill in the line. The remaining lines move up one line without change. The new text is inserted prior to the remainder of the next line. Undo this change.

51 Move the cursor to the N in the next line.

52 Temporarily change the previous sentence by typing the following and pressing the GO key:

```
c(xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxxGO
```

After typing c(, the previous sentence is deleted. The remainder of the file moves up as in the last change. The new text is inserted before the next sentence. Undo this change.

53 Move the cursor to the A in the third sentence.

54 Temporarily change the next three sentences by typing the following and pressing the GO key:

```
c3)xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

After typing **c3)**, the next three sentences are deleted. "You can" fills in the line the cursor is on. The remainder of the file moves up without change. The new text is inserted before "You can." Undo this change.

You can change more than one previous sentence in the same manner using **c(** with the number of sentences before or after the **c**.

55 Move the cursor to the s in the word "simply" in the third sentence.

56 Temporarily change the remainder of this sentence by typing the following:

```
c)xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Beginning with the word "simply," the remainder of the sentence is deleted. The remainder of the next line fills in the line the cursor is on. The remaining portion of the file moves up without change. The new text is inserted before "Normally." Undo this change. You can change more than one sentence in this manner in both the forward and reverse directions.

57 Move the cursor to the first line of the second paragraph.

58 Temporarily change this paragraph by typing the following and pressing the GO key:

```
c}xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The second paragraph is deleted. The next two paragraphs are moved up with three blank lines between the first and second paragraphs. The cursor is at the beginning of the second empty line. The new paragraph is inserted between the first and second paragraphs. Undo this change.

59 Move the cursor to the empty line before paragraph 2.

60 Change the previous paragraph by typing the following and pressing the GO key:

```
c{xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The first paragraph is deleted. The cursor is on a blank line that is now the first line in the file. This line is followed by another blank line. The new text is inserted as the first paragraph. Undo this change.

61 With the cursor on the first line in the file, change the first two paragraphs by typing the following and pressing the GO key:

```
c2}xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The first two paragraphs are deleted. The cursor is on a blank line that is now the first line in the file.

This line is followed by another blank line. The new text is inserted as the first paragraph. Additional paragraphs can be added by continuing to type text in a new paragraph. Undo this change.

62 Move the cursor to the space before the third paragraph.

63 Change the first two paragraphs by typing the following and pressing the GO key:

```
c2{xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The first two paragraphs are deleted. The cursor is on a blank line that is now the first line in the file. This line is followed by another blank line. The new text is inserted as the first paragraph. Additional paragraphs can be added by continuing to type text in a new paragraph. Undo this change.

64 Move the cursor to the A in the third sentence in the first paragraph.

65 Change the remainder of paragraph one by typing the following and pressing the GO key:

```
c}xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The remainder of the paragraph is deleted. Paragraphs two through four move up with no change. A blank line remains between paragraphs one and two. The new text is added on to the end of the first paragraph. Undo this change.

You can change parts of paragraphs in both the forward and reverse directions. You can also change more than one paragraph using this method.

66 Move the cursor to the s in successfully in the first line of the first paragraph.

67 Change all information from the cursor location up to the "A" in the fifth line by typing the following and pressing the GO key:

```
c/A (Press the RETURN key)
xxxxxxxxType any number of linesxxxxxx
(Press the RETURN key)
xxxxxxxxxxxxxxxxxxxxxxxxxxx
```

All the information from the space after "have" up to "A" is deleted. Beginning with "A", the remainder of the fifth line fills in line one. The remainder of the file moves up with no change. The new text is added preceding the A. Undo this change.

68 Save this file for use in the next example. The file should appear as follows:

```
When you have successfully logged in, a program
called the shell is listening to your terminal.
The shell reads the line you type, splits it into
a command name and its arguments, and executes the
command.  A command is simply an executable program
located in a file.  Normally, the shell looks first
in your current directory for a program with your
command name, and if none is there, then in system
directories.  There is nothing special about
system-provided commands except that they are kept
in directories where the shell can find them.  You can
also keep commands in your own directories and arrange
for the shell to find them there.

The command name is the first word on an input line
to shell; the command and its arguments are separated
from one another by space and/or tab characters.

When a program terminates, the shell will ordinarily
regain control and type a $ at you to indicate that
it is ready for another command.  The shell has many
other capabilities described in detail in
the user's manual under sh.

The file system is arranged in a hierarchy of
directories.  When the system administrator gave you
a user name, he or she also created a directory
for you (ordinarily with the same name as your user
name, and known as your login or home directory).
When you log in, that directory becomes your current
directory, and any file name you type is by default
assumed to be in that directory.  Because you are the
owner of this directory, you have full permissions
to read, write, alter, or destroy its contents.
Permissions to have your will with other directories
and files will have been granted or denied to you by
their respective owners or by the system administrator.
```

Summary

~	Switches character from lowercase to uppercase and vise versa.
C	Changes from cursor to the end of the line (same as **c$**).
S	Substitute for or change a whole line (same as **cc**).
c[*object*]	Change the specified [*object*] to the following text until the GO key is pressed.
r	Replaces a character.
s	Replaces a character with a string until the GO key is pressed.
cc	Changes a whole line.

Copying Text

The Concept of Yank and Put

Vi provides a method of making a copy of text and placing this copy in another location in the file. This method is called "yank and put." The **y** operator yanks a copy of any specified object (word, line, sentence, or paragraph) into a specially reserved space called a register. The text can then be put back in the file from the register with the commands **p** and **P**; the **p** command puts the text after or below the cursor while **P** puts the text before or above the cursor.

If the text you yank forms a part of a line or is an object such as a sentence that partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if you use **P**). If the yanked text forms whole lines, whole lines will be put back without changing the current line.

The **Y** command is used to make a copy of a line. The cursor can then be moved to any character on another line and the **p** used to place the yanked line following the current line. **P** places the copied line above the current line. The **YP** command makes a copy of the current line and places it before the current line. The cursor is placed on the first character of this copy. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** will also make a copy of the current line and place it after the current line. You can give **Y** a count of lines to yank and thus duplicate several lines.

Vi has a single unnamed register where the last yanked text is saved. Each time a yank command is performed that uses the unnamed register, the previous yank command is lost. To prevent the loss of this text, the editor has a set of named registers **a** through **z** that can be used to save copies of text. The general format of the **yank** command using named registers is

"xyobject

where *x* is the name of the register (**a** through **z**) into which an object is copied. The following procedure copies a line into a new location in a file.

1 Enter

```
"ayy
```

This yanks a line from where the cursor is into the named register **a**.

2 Move the cursor to the eventual resting place of this line

3 Enter

```
"ap or
"aP
```

This puts the line at the new location.

Copying Objects

Yank and put can be used to copy the following objects:

- Characters.
- Words.
- Lines.
- Sentences.
- Paragraphs.

Note: *Each of the following objects should be experimented with to fully understand what happens during a yank and put.*

Characters can be copied by typing the **yank** command and then typing one of the following object commands:

SPACEBAR	Yanks one character in forward direction.
BACKSPACE	Yanks one character in backward direction.
h	Yanks one character in backward direction.
l	Yanks one character in forward direction.
f*x*	Yanks all characters from cursor up to *x* in forward direction.
F*x*	Yanks all characters from cursor up to *x* in backward direction.
t*x*	Yanks all characters from cursor up to and including *x* in forward direction.
T*x*	Yanks all characters from cursor up to and including *x* in backward direction.

Words can be copied by typing the **yank** command and then typing one of the following objects:

w	Yanks one word in forward direction (punctuation counts as word).
W	Yanks one word in forward direction (punctuation does not count as word).
b	Yanks one word in backward direction (punctuation counts as word).
B	Yanks one word in backward direction (punctuation does not count as word).
e	Yanks one word in forward direction up to last character in word (punctuation counts as word).

Lines can be copied (in addition to **yy** and **Y**) by typing the **yank** command and then typing one of the following objects:

$	Yanks one line from cursor to end of line.
RETURN	Yanks one line plus line cursor is on in forward direction.
j	Yanks one line plus line cursor is on in forward direction.
+	Yanks one line plus line cursor is on in forward direction.
k	Yanks one line plus line cursor is on in backward direction.
-	Yanks one line plus line cursor is on in backward direction.
H	Yanks line cursor is on through top line on screen.
M	Yanks line cursor is on through middle line on screen.
L	Yanks line cursor is on through bottom line on screen.
G	Yanks line cursor is on through last line in file. If a number precedes **G**, yanks through that line in forward or reverse direction.
/	Yanks from where cursor is up to "searched for"string in forward direction.
?	Yanks from where cursor is through "searched for"string in backward direction.

Sentences can be copied by typing the yank command and then typing one of the following objects:

)	Yanks from cursor to end of sentence in forward direction.
(	Yanks from cursor to beginning of sentence in reverse direction.

Paragraphs can be copied by typing the **yank** command and then typing one of the following objects:

}	Yanks from cursor to end of paragraph in forward direction.
{	Yanks from cursor to beginning of paragraph in reverse direction.

All of the object commands shown above can be preceded by a number. Using a number allows you to copy more than one of the object. This is especially useful when copying characters. The following exercise will demonstrate only a few of the possible ways to copy text.

Example Six

Note: *This example contains information covered in Example Two. The information in Example Two should be reviewed prior to performing this example. Only a selected amount of text objects that can be copied is covered in this example.*

1 Copying characters.

a Make a copy of the file shown at the end of Example Five.

b Access your file by entering the following command and pressing the RETURN key:

```
$ vi filename
```

The cursor is on the first character of the first line in the file.

c Yank characters up to the first f on line 1 by typing the following:

```
ytf
```

Nothing appears to happen. The editor has yanked the desired characters into the unnamed register.

d Move the cursor to the open line between the first and second paragraphs.

e Put the characters just yanked into the unnamed register by typing a lowercase **p**.

```
p
```

If you had success, "When you have success" appears on the blank line.

f Move the cursor to the first character in the first line in the third paragraph.

g Type a lowercase **p**.

```
p
```

"When you have success" is put after the "W" in the line you are on.

h Move the cursor to the first character in the first line of the fourth paragraph.

i Type an uppercase **P**.

```
P
```

"When you have success" is put before the "T" in the line you are on.

j Move the cursor to the end of the last line in the third paragraph.

k Type a lowercase **p**.

```
p
```

"When you have success" is added to the end of the line.

l Undo the last put by typing a lowercase **u**.

```
u
```

"When you have success" is deleted from the end of the line.

m Move the cursor back to the first character in the fourth paragraph.

n Type a lowercase u.

```
u
```

"When you have success" is added back to the end of paragraph three.

The line you are on cannot now be undone.

o Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

p Access your file again.

```
$ vi samefile
```

The file does not contain your yank and put changes.

q With the cursor on the first character in the first paragraph, type the following and press the SPACEBAR:

```
y
```

Nothing happens. You have yanked the W into the unnamed register.

r Move the cursor to the next line.

s Type a lowercase **p**.

```
p
```

The W follows the c in the second line. Not too useful.

t Move the cursor to the first character in the first paragraph.

u Yank the first 21 characters by typing the following:

```
y21 (Press the SPACEBAR)
```

Nothing happens. The 21 characters have been yanked into the unnamed register.

v Move the cursor to the space between the first and second paragraphs.

w Perform a put by typing a lowercase **p**.

```
p
```

"When you have success" is put on the empty line.

x Move the cursor to the beginning of the third line in the first paragraph.

y Yank the first 15 characters into a named register by typing the following:

```
"ay15l
```

Nothing appears to happen. The editor has yanked the first 15 characters on the line into the register a.

z Move the cursor to the space between the second and third paragraph.

aa Put the information in the named register a on the empty line by typing the following:

```
"ap
```

"The shell reads" is put on the empty line.

ab Move the cursor to the space after name in the fourth line of the first paragraph.

ac Yank to the beginning of the line by typing the following:

```
y14 (Press the BACKSPACE key)
```

Nothing appears to happen. The 14 characters have been yanked into the unnamed register.

ad Move the cursor to the empty line between the third and fourth paragraphs.

ae Put the information in the unnamed register on the empty line by typing a lowercase **p**.

```
p
```

"a command name" is put on the empty line.

af Undo the last put by typing a lowercase **u**.

```
u
```

A blank line reappears.

ag Put the information in the named register on the empty line by typing the following:

```
"ap
```

"The shell reads" is put on the empty line.

ah Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

ai Access your file again.

```
$ vi samefile
```

The file does not contain your yank and put changes.

aj Move your cursor to the space between the first and second paragraph.

ak Attempt to enter information in register a by typing the following:

```
"ap
```

At the bottom of the screen the editor prints nothing in register a.

When you quit or write and quit a file, the information in the named or unnamed registers is lost.

2 Copying words.

a Move your cursor to the s in successfully in the first line of the file.

b Yank a copy of this word by typing the following:

```
yw
```

Nothing appears to happen. The word "successfully" is yanked into the unnamed register.

c Move the cursor to the empty line between the first and second paragraph.

d Put the word in the empty line by typing a lowercase **p**.

```
p
```

"successfully" is copied from the unnamed register into the empty line.

e Move the cursor to the space before terminates in the first line of the third paragraph.

f Put the word in the unnamed register before terminates by typing a lowercase **p**.

```
p
```

"successfully" is inserted before terminates.

g Undo this put by typing a lowercase **u**.

```
u
```

"successfully" disappears.

h The cursor is now on the t in terminates. Put the word before terminates by typing an uppercase **P**.

```
P
```

"successfully" is inserted before terminates.

i Move the cursor to the "t" in "the" in the second line of the first paragraph.

j Yank a copy of the next four words beginning with "the" by typing the following:

```
"by4w
```

Nothing appears to happen. "the shell is listening" is placed in the named register b.

k Move the cursor to the "i" in "is" in the first line of the fourth paragraph.

l Put the four words in the b register before "is" by typing the following:

```
"bP
```

"the shell is listening" is put between "system" and "is."

m Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

3 Copying lines.

a Access your file again.

```
$ vi samefile
```

The file does not contain your yank and put changes.

b With the cursor on any character in the first paragraph, type an uppercase **Y**.

```
Y
```

The first line has been copied into the unnamed register.

c Move the cursor to any character in the fifth line of the first paragraph.

d Put a copy of the first line below this line by typing a lowercase **p**.

```
p
```

e Move the cursor to any character in the last line of the first paragraph.

f Put the line in the unnamed register before the current line by typing an uppercase **P**.

```
P
```

g Move the cursor to any character in the first line in the second paragraph.

h Type the following:

```
YP
```

The current line is duplicated. The cursor is on the first character of the top copy.

i Move the cursor to any character in the first line of the third paragraph.

j Yank a copy of three lines into named register g by typing the following:

```
"g3Y
```

Nothing appears to happen. The first three lines in the third paragraph are in register g.

k Move the cursor to any character in the second line of the fourth paragraph.

l Put the information in register g to follow the second line by typing the following:

```
"gp
```

Three lines from paragraph 3 are copied below the second line.

m Attempt to put information from register h to follow the second line by typing the following:

```
"hp
```

Nothing in register h is displayed at the bottom of the screen. Only register g of the named registers has information stored.

n Move the cursor to the "l" in "line" in the third line in the first paragraph.

o Yank a copy of the remainder of the line by typing the following:

```
y$
```

Nothing appears to happen. From "line" to the end of the line is in the unnamed register.

p Move the cursor to the first character in the last line in the first paragraph.

q Copy the information in the unnamed register at the beginning this line by typing an uppercase **P**.

```
P
```

The characters on the line move right to follow the information put from the unnamed register.

r Move the cursor to any character in the first line of the file.

s Yank eight lines by typing the following:

```
y8$
```

At the bottom of the screen appears the message "8 lines yanked."

t Move the cursor to any character in any line of the file.

u Put a copy of the first eight lines at your present location by typing the following:

```
p
```

At the bottom of the screen appears the message "8 more lines." The eight lines from the top of the file follow the line the cursor is on.

v Undo the last put by typing a lowercase **u**.

```
u
```

At the bottom of the screen appears the message "8 fewer lines." The eight new lines disappear.

w Move the cursor to the "s" in "successfully" in the first line of the file.

x Yank from "successfully" up to the word "simply" in the fifth line of the paragraph by typing the following:

```
y/simply
```

At the bottom of the file appears five lines yanked.

y Move the cursor to the space following "terminates" in the first line of the third paragraph.

z Place a copy of the yanked five lines here by typing a lowercase **p**.

```
p
```

The information from "successfully" through the "is" preceding "simply" is copied between "terminates" and "the."

Normally, when the editor does something with five or more lines, an indication is provided at the bottom of the screen. The editor looked at the put as four lines even though it yanked five lines.

When yanking parts of a line accompanied by one or more lines into the unnamed register, the yank can only be put at another location once. A yank of this kind, where the yanked text is to be put in more than one location, must be placed in a named register.

aa Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

4 Copying sentences.

a Access your file again.

```
$ vi samefile
```

The file does not contain your yank and put changes.

b Move the cursor to the first character in the third sentence of the first paragraph.

Assure that two spaces are at the end of this sentence.

c Yank this sentence by typing the following:

```
y)
```

Nothing appears to happen. A copy of the sentence is in the unnamed register.

d Move the cursor to the space preceding "There" in the ninth line of the first paragraph.

e Put the sentence before "There" by typing a lowercase **p**.

```
p
```

The sentence is inserted between "directories" and "There." When yanking parts of a line accompanied by one or more lines into the unnamed register, the yank can only be put at another location once. A yank of this kind, where the yanked text is to be put in more than one location, must be placed in a named register.

f Move the cursor to the "W" in "When" in the second line of the fourth paragraph.

g Yank a copy of the next two sentences in the named register d by typing the following:

```
"dy2)
```

At the bottom of the screen appears the message "7 lines yanked." The two sentences are in register d.

h Move the cursor to the space before "The" in the third line of the third paragraph.

i Insert the two sentences in register d before "The" by typing the following:

```
"dp
```

At the bottom of the screen appears the message "6 more lines." The two sentences are put between "command" and "The."

j Move the cursor to the end of the second line in the first paragraph.

k Put the two sentences in register d after this sentence by typing the following:

```
"dp
```

At the bottom of the screen appears the message "6 more lines." The two sentences are added following the end of the line. The line the cursor is on wraps around to what appears to be another line. This line can be split by using a delete and put covered in the subsection "Moving Text."

l Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

5 Copying paragraphs.

a Access your file again.

```
$ vi samefile
```

The file does not contain your yank and put changes.

b With the cursor on the first character in the file, yank the first paragraph into the unnamed register by typing the following:

```
y}
```

At the bottom of the screen appears the message "13 lines yanked." The first paragraph is copied into the unnamed register. Move the cursor to any character in the last line of the file.

c Put a copy of the first paragraph after the last paragraph by typing a lowercase p.

```
p
```

At the bottom of the screen appears the message "13 more lines." Paragraph one is copied following the last paragraph. A space can be added by typing an uppercase **O** and pressing the GO key.

d Move the cursor to space between the first and second paragraph.

e Yank a copy of the next two paragraphs by typing the following:

```
y2}
```

At the bottom of the screen appears the message "10 lines yanked." The next two paragraphs are in the unnamed register.

f Move the cursor to any character in the last line of the third paragraph.

g Add the two yanked paragraphs after the current paragraph by typing a lowercase **p**.

```
p
```

At the bottom of the screen appears the message "10 more lines." The two paragraphs are added with the required space between paragraphs.

Spacing between paragraphs may not be yanked correctly always. A space can be added by typing an uppercase **O** and pressing the GO key.

h Move the cursor to the "N" in "Normally" in the sixth line of the first paragraph.

i Yank a copy of the remainder of this paragraph and the next paragraph and place them in the named register a by typing the following:

```
"ay2}
```

At the bottom of the screen appears the message "12 lines yanked." From "Normally" through the end of the next paragraph is in register a.

j Move the cursor to space before "The" in the third line of the third paragraph.

k Put the text in register a by typing the following:

```
"ap
```

At the bottom of the screen appears the message "11 more lines." The partial paragraph and the next paragraph is inserted between "command." and "The."

l Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

m Save this file for later use.

Summary

"a-"z	Named registers that can precede a yank command.
p	Puts yanked text after or below the cursor.
P	Puts yanked text before or above the cursor.
y	Yanks a copy of the following object into a register.
Y	Yanks a copy of the current line into a register.
yy	Same as Y. (Yanks a copy of the current line into a register.)

Moving Text

To move text from one location to another, use the following procedure:

1 Delete (or change) the information you need to move; it will be saved in an area designated as a register.

2 Move the cursor to the location you wish to copy the information just deleted and perform a "put."

The delete operation may be any form of a delete such as:

```
x
dd
d[object]
C
c[object]
and so on
```

The text can be put back in the file with the commands **p** or **P**; the **p** command puts the text after or below the cursor while **P** puts the text before or above the cursor. An example of a **delete-and-put** command is:

```
xp
```

The **x** deletes the character the cursor is on; the cursor moves to the next character to the right. The **p** puts the deleted character back following the character the cursor is on. The result is two characters have swapped positions.

If the text you delete forms a part of a line or is an object such as a sentence that partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if you use **P**). If the deleted text forms whole lines, they will be put back as whole lines without changing the current line.

You may wish to place the text you are to move into a specific location. The editor has a set of named registers a through z that you can use to save copies of text. The general format of the **delete** command using named registers is

```
"xdelete object
or
"xchange object
```

where *x* is the name of the register (a through z) into which an object is deleted. The following procedure moves a line to a new location in a file.

1 Enter

```
"add
```

This deletes the line the cursor is on into the named register a.

2 Move the cursor to the eventual resting place of this line.

3 Enter

```
"ap or
"aP
```

This puts the line at the new location. You can also do the same with a change operation. After the new text is entered and the escape key operated, the deleted text can be "put" at another location in the file.

Moving Objects

Any **delete** command or **change** command can be used to move the deleted text to a new location in your file. The following objects can be moved in your file:

- Characters.
- Words.
- Lines.
- Sentences.
- Paragraphs.

When deleting or changing, each of the deleted objects is automatically placed in the unnamed register unless preceded by the named register command. The **delete** and **change** commands are covered in earlier parts of this section and will not be covered here.

Example Seven

This exercise contains information covered in Examples Two, Four, and Five. The information in these exercises should be reviewed prior to performing this examples. Only a selected amount of text objects that can be moved is covered in this example.

1 Moving characters.

a Make a copy of the file shown at the end of Example Five.

b Access your file by typing

```
$ vi filename
```

The cursor is on the first character of the first line in the file.

c Switch the first two characters by typing the following:

```
xp
```

The "W" was deleted and put into the unnamed register by the "y." The cursor moved to the next character and the "W" put back after the cursor from this register by the "p."

d With the cursor on the first character in this line, delete characters up to the first f on the first line by typing the following:

```
dtf
```

Characters are deleted. The editor has deleted the desired characters into the unnamed register.

e Move the cursor to the open line between the first and second paragraphs.

f Put the characters just deleted on this line by typing a lowercase **p**.

```
p
```

If you had success, "hWen you have success" appears on the blank line.

g Move the cursor to the first character in the first line in the third paragraph.

h Type a lowercase **p**.

```
p
```

"hWen you have success" is put after the "W" in the line you are on.

i Move the cursor to the first character in the first line of the fourth paragraph.

j Type an uppercase **P**.

```
P
```

"hWen you have success" is put before the "T" in the line you are on.

k Move the cursor to the end of the last line in the third paragraph.

l Type a lowercase **p**.

```
p
```

"hWen you have success" is added to the end of the line.

m Undo the last put by typing a lowercase **u**.

```
u
```

"hWen you have success" is deleted from the end of the line.

n Move the cursor back to the the first character in the fourth paragraph.

o Type a lowercase **u**.

```
u
```

"hWen you have success" is added back to the end of paragraph three.

The line you are on cannot now be undone.

p Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

q Access your file again.

```
$ vi samefile
```

The file does not contain your delete and put changes.

r With the cursor on the second character in the first paragraph, type the following command and press the SPACEBAR:

```
d
```

The W is deleted. You have deleted the W into the unnamed register.

s Move the cursor to the next line.

t Type a lowercase **p**.

```
p
```

The W follows the c in the second line. Not too useful.

u Move the cursor to the first character in the first paragraph.

v Delete the first 20 characters by typing the following and pressing the SPACEBAR:

```
d20
```

The characters are deleted. The 20 characters have been deleted into the unnamed register.

w Move the cursor to the space between the first and second paragraphs.

x Perform a put by typing a lowercase **p**.

```
p
```

"hen you have success" is put on the empty line.

y Move the cursor to the beginning of the third line in the first paragraph.

z Delete the first 15 characters into a named register by typing the following:

```
"ad15l
```

The characters are deleted. The editor has deleted the first 15 characters on the line into the register a.

aa Move the cursor to the space between the second and third paragraph.

ab Put the information in the named register a on the empty line by typing the following:

```
"ap
```

"The shell reads" is put on the empty line.

ac Move the cursor to the space after name in the fourth line of the first paragraph.

ad Delete to the beginning of the line by typing the following and pressing the BACKSPACE key:

```
d14
```

The characters are deleted. The 14 characters have been deleted into the unnamed register.

ae Move the cursor to the empty line between the third and fourth paragraphs.

af Put the information in the unnamed register on the empty line by typing a lowercase **p**.

```
p
```

"a command name" is put on the empty line.

ag Undo the last put by typing a lowercase **u**.

```
u
```

A blank line reappears.

ah Put the information in the named register on the empty line by typing the following:

```
"ap
```

"The shell reads" is put on the empty line.

ai Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

aj Access your file again.

```
$ vi samefile
```

The file does not contain your delete and put changes.

ak Move your cursor to the space between the first and second paragraph.

al Attempt to enter information in register a by typing the following:

```
"ap
```

At the bottom of the screen, the editor prints nothing in register a.

When you quit or write and quit a file, the information in the named or unnamed registers is lost.

2 Moving words.

a Move your cursor to the s in successfully in the first line of the file.

b Delete this word by typing the following:

```
dw
```

"successfully" is deleted. The word "successfully" is deleted into the unnamed register.

c Move the cursor to the empty line between the first and second paragraph.

d Put the word in the empty line by typing a lowercase **p**.

```
p
```

"successfully" is copied from the unnamed register into the empty line.

e Move the cursor to the space before "terminates" in the first line of the third paragraph.

f Put the word in the unnamed register before "terminates" by typing a lowercase **p**.

```
p
```

"successfully" is inserted before "terminates."

g Undo this put by typing a lowercase **u**.

```
u
```

"successfully" disappears.

h The cursor is now on the "t" in "terminates." Put the word before terminates by typing an uppercase **P**.

```
P
```

"successfully" is inserted before "terminates."

i Move the cursor to the "t" in "the" in the second line of the first paragraph.

j Delete the next four words beginning with "the" and put them into register b by typing the following:

```
"bd4w
```

The four words are deleted. "the shell is listening" is placed in the named register b.

k Move the cursor to the "i" in "is" in the first line of the fourth paragraph.

l Put the four words in the b register before "is" by typing the following:

```
"bP
```

"the shell is listening" is put between "system" and "is."

m Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

3 Moving lines.

a Access your file again.

```
$ vi samefile
```

The file does not contain your delete and put changes.

b With the cursor on any character in the first line of the first paragraph, delete this line by typing the following:

```
dd
```

The first line has been deleted and copied into the unnamed register.

c Move the cursor to any character in the fourth line of the first paragraph.

d Put a copy of the first line below this line by typing a lowercase **p**.

```
p
```

e Move the cursor to any character in the last line of the first paragraph.

f Put the line in the unnamed register before the current line by typing an uppercase **P**.

```
P
```

The deleted first line is placed before the current line.

g Move the cursor to any character in the first line of the third paragraph.

h Delete this line and the next two lines into named register g by typing the following:

```
"g3dd
```

Three lines are deleted. The first three lines in the third paragraph are in register g.

i Move the cursor to any character in the second line of the fourth paragraph.

j Put the information in register g to follow the second line by typing the following:

```
"gp
```

Three lines from the third paragraph are copied below the second line.

k Attempt to put information from register h to follow the second line by typing the following:

```
"hp
```

Nothing in register h is displayed at the bottom of the screen. Only register g of the named registers has information stored.

l Move the cursor to the "l" in "line" in the second line in the first paragraph.

m Delete the remainder of the line by typing the following:

```
d$
```

The remainder of the line is deleted. From "line" to the end of the line is in the unnamed register.

n Type a lowercase **o** and press the GO key.

```
o
```

An empty line is created.

o Put the remainder of the previous line here by typing a lowercase **p**.

```
p
```

You have just split a line. This is especially useful when a line is too long (sometimes wrapping to the next line).

An easier way to split a line is to move the cursor to the space before the first character of the word where the split is to occur and type **r**. Then press the RETURN key.

p Move the cursor to the first character in the last line in the first paragraph.

q Put the information in the unnamed register before this line, but on the same line by typing an uppercase **P**.

```
P
```

The characters on the line move right to follow the information put from the unnamed register.

r Move the cursor to any character in the first line of the file.

s Delete eight lines by typing the following:

```
d8$
```

At the bottom of the screen appears the message "8 lines deleted."

t Move the cursor to any character in any line of the file.

u Put a copy of the first eight lines at your present location by typing the following:

```
p
```

At the bottom of the screen appears message "8 more lines." The eight lines from the top of the file follow the line the cursor is on.

v Undo the last put by typing a lowercase u.

```
u
```

At the bottom of the screen appears the message "8 fewer lines." The eight new lines disappear.

w Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

x Access your file again.

```
$ vi samefile
```

y Move the cursor to the "s" in "successfully" in the first line of the file.

z Delete from "successfully" up to the word "simply" in the fifth line of the paragraph by typing the following:

```
d/simply
```

At the bottom of the file appears the message "5 lines deleted."

aa Move the cursor to the space following "terminates" in the first line of the third paragraph.

ab Place a copy of the deleted five lines here by typing a lowercase **p**.

```
p
```

The information from "successfully" through the "is" preceding "simply" is copied between "terminates" and "the."

Normally, when the editor does something with five or more lines, an indication is provided at the bottom of the screen. The editor looked at the put as four lines even though it deleted five lines.

When deleting parts of a line accompanied by one or more lines into the unnamed register, the delete can only be put at another location once. A delete of this kind, where the deleted text is to be put in more than one location, must be placed in a named register.

ac Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

4 Moving sentences.

a Access your file again.

```
4 vi samefile
```

The file does not contain your delete and put changes.

b Move the cursor to the first character in the third sentence of the first paragraph. Make sure that two spaces are at the end of this sentence.

c Delete this sentence by typing the following:

```
d)
```

The sentence is deleted. A copy of the sentence is in the unnamed register.

d Move the cursor to the space preceding "There" in the seventh line of the first paragraph.

e Put the sentence before "There" by typing a lowercase **p**.

```
p
```

The sentence is inserted between "directories" and "There."

When deleting parts of a line accompanied by one or more lines into the unnamed register, the delete can only be put at another location once. A delete of this kind, where the deleted text is to be put in more than one location, must be placed in a named register.

f Move the cursor to the "W" in "When" in the second line of the fourth paragraph.

g Delete the next two sentences into the named register m by typing the following:

```
"md2)
```

At the bottom of the screen appears the message "7 lines deleted." The two sentences are in register m.

h Move the cursor to the space before "The" in the third line of the third paragraph.

i Insert the two sentences in register m before "The" by typing the following:

```
"mp
```

At the bottom of the screen appears the message "6 more lines. The two sentences are put between "command" and "The."

j Move the cursor to the end of the second line in the first paragraph.

k Put the two sentences in register m after this sentence by typing the following:

```
"dp
```

At the bottom of the screen appears the message "6 more lines." The two sentences are added following the end of the line.

The line the cursor is on wraps around to what appears to be another line. This line can be split by using a delete and put. Refer to moving lines in this example.

l Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

5 Moving paragraphs.

a Access your file again.

```
$ vi samefile
```

The file does not contain your delete and put changes.

b With the cursor on the first character in the file delete the first paragraph into the unnamed register by typing the following:

```
d}
```

At the bottom of the screen appears the message "13 lines deleted." The first paragraph is copied into the unnamed register. Move the cursor to any character in the last line of the file

c Put a copy of the first paragraph after the last paragraph by typing a lowercase **p**.

```
p
```

At the bottom of the screen appears the message "13 more lines." Paragraph one is copied following the last paragraph.

A space can be added by typing an uppercase **O** and pressing the GO key.

d Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

e Access your file again.

```
$ vi samefile
```

The file does not contain your delete and put changes.

f Move the cursor to space between the first and second paragraph.

g Delete the next two paragraphs by typing the following:

```
d2}
```

At the bottom of the screen appears the message "10 lines deleted." The next two paragraphs are in the unnamed register.

h Move the cursor to any character in the last line of the first paragraph.

i Add the two deleted paragraphs after the current paragraph by typing a lowercase **p**.

```
p
```

At the bottom of the screen appears the message "10 more lines." The two paragraphs are added with the required space between paragraphs.

Spacing between paragraphs may not be moved correctly always. A space can be added by typing an uppercase or lowercase **o** and pressing the GO key.

j Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

k Access your file again.

```
$ vi samefile
```

The file does not contain your delete and put changes.

l Move the cursor to the "N" in "Normally" in the sixth line of the first paragraph.

m Delete the remainder of this paragraph and the next paragraph and place them in the named register a by typing the following:

```
"ad2}
```

At the bottom of the screen appears the message "12 lines deleted." From "Normally" through the end of the next paragraph is in register a.

n Move the cursor to space before "The" in the third line of the third paragraph.

o Put the text in register a here by typing the following:

```
"ap
```

At the bottom of the screen appears the message "11 more lines." The partial paragraph and the next paragraph is inserted between "command." and "The."

p Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

6 Moving objects using the **change** command.

a Access your file again.

```
$ vi samefile
```

The file does not contain your delete and put changes.

b With the cursor on any character in the first line of the file, change the first line as follows:

```
cc
```

The first line disappears; type in some x's and then press the GO key:

```
xxxxxxxxxxxxx
```

The deleted line has been put into the unnamed buffer.

c Move the cursor to any character in the fifth line of the first paragraph.

d Put the line just deleted by the change command after this line by typing a lowercase **p**.

```
p
```

The deleted line is added after the current line.

e Move the cursor to any character in the first line of the second paragraph.

f Change the next two lines and put them in register a by typing the following:

```
"a2C
```

The two lines are deleted; add some x's and press the GO key:

```
xxxxxxxxxxxxx
```

The deleted lines have been put into the named buffer a.

g Move the cursor to any character in the first line of the last paragraph.

h Put the two lines in register a before the line you are on by typing the following:

```
"aP
```

The two lines deleted by the previous change are added above the fourth paragraph.

i Exit your file without writing the changes just performed by typing the following and pressing the RETURN key:

```
:q!
```

j **Change** commands are similar to **delete** commands except text is added to replace the deleted text. Repeat this exercise substituting the **change** command and text for the **delete** command.

k There are many other ways to move text using the delete or change operators and object symbols. Experiment and learn.

l Save this file for later use.

Summary

"a-"z	**Named registers that can precede a delete command.**
p	**Puts deleted text after or below the cursor.**
P	**Puts deleted text before or above the cursor.**

Repeating Actions with the . Command

Vi provides a time saving command, called the "dot" command. You can repeat the last action by placing the cursor at the location you wish to repeat the last action and typing a:

The following actions can be repeated using the dot (.) command.

- Append.
- Insert.
- Open.
- Delete.
- Change.
- Put.

The following paragraphs illustrate how each of the above actions can be repeated using the (.) command.

When you append text at a location and press the GO key, it may be necessary to add this same text to another location. In this case, move the cursor to the character that you desire the text to follow and type a dot.

When you insert text and press the GO key, move the cursor to the character that you desire the text to precede and type a dot. When you have opened a new line, added new text, and pressed the GO key, move the cursor to the line where the new text is to be repeated and one of the following will occur when you type a dot. If you had opened the original line using a lowercase **o**, the repeated text will be added following the line your cursor is on. If you had opened the original line using an uppercase **O**, the repeated text will be added preceding the line your cursor is on.

After you have deleted text by using an **x**, **dd**, **D** or a delete with an **object** command, move the cursor to another location; and by typing a dot, repeat this same command. For example, if you give the command delete two words by typing **d2w**, the two words you desire to remove are deleted. You can now move the cursor to the first character of two different words and type a dot and the two new words will be deleted.

After you have changed text by using an **r**, **s**, **S**, , **cc**, **C**, or a **change** with an **object** command, move the cursor to another location; and by typing a dot, place the same text added by the previous **change** command. For example, if you give the command change two lines by typing **2cc** and type in some text, the line you are on and the next line is replaced by the new text. You can now move your cursor to another line and type a dot. The line your cursor is on and the next line will be replaced by the text added in the last change.

After you have executed a **put** command that is associated with the unnamed register and if you desire to place this text at another location, the text in the unnamed register may not be the same. In most cases, you cannot use the dot (.) command to place the original text at the new location.

If you desire to place text at another location that is in a named register after doing a put, you can save time by using the dot (.) command instead of having to repeat the register name.

Example Eight

1 Access the file containing the text at the end of Example Five:

```
$ vi filename
```

2 Move the cursor to the "n" in "When" in the first line of the first paragraph.

3 Append to this word by typing an a followed by a space, five x's, and press the GO key:

```
a xxxxx
```

Five x's now follow When.

4 Move the cursor to the "d" in "located" in the sixth line of the first paragraph. Append the five x's by typing the following:

```
.
```

The five x's just added could be a word you wish to repeat.

5 Move the cursor to the "d" in "command" in the eighth line of the first paragraph and type the following:

```
.
```

The five x's are appended for the second time.

6 Move the cursor to the "e" in "except" in the tenth line of the first paragraph.

7 Insert two lines of y's by typing the following and pressing the GO key:

```
iyyyyy (Press the RETURN key)
yyyyy
```

7 The line is split with five y's at the end of the first line and five y's preceding the second line.

8 Move the cursor to the "t" in "them" in the last line of the first paragraph.

9 Insert the two lines of y's by typing the following:

```
.
```

The line splits with five y's at the end of the first line and five y's preceding "them" on the second line.

10 Move the cursor to any character in the first line of the second paragraph.

11 Add a line of text to follow this line by typing the following and pressing the GO key:

```
ozzzzzzzzzz
```

A blank line opens after the first line and ten z's are added to this line.

12 Move the cursor to any character in the first line of the third paragraph.

13 Add the ten z's by typing the following:

The ten z's are added on a new line following the line the cursor is on.

14 Delete the ten z's on this line by typing the following:

```
dd
```

The z's disappear and the lines close up.

15 Move the cursor to the last line in the third paragraph.

16 Delete this line by using the dot (.) command.

```
.
```

The last line disappears and the file closes up.

17 Move the cursor the "d" in "directory" in the third sentence of the fourth paragraph.

18 Change this word to "location" by typing the following and pressing the GO key:

```
cwlocation
```

19 Locate each of the remaining occurrences of "directory" in this paragraph and by using the dot (.) command change the word to "location."

Note that the cursor must be placed on the "d" to make the correct change. Observe that another change must be made to change "directories."

20 Move the cursor to the "B" in "Because" in the eighth line in the fourth paragraph.

21 Yank the remainder of this line and the next two lines by typing the following:

```
y3$
```

22 Move the cursor to the space before "The" in the third paragraph.

23 Insert the three lines just yanked by typing a lowercase **p**.

```
p
```

24 Move the cursor to the space before "You" near the end of the first paragraph.

25 Attempt to enter the three lines added in the previous step by typing a dot.

```
.
```

Wrong text. The same thing would have happened if you had entered another **p**. This is a good reason for using a named register. When using a named register, the text can be copied in a number of locations using the . or a **"xp**. The . can be used only to repeat the last **"xp**.

26 Exit your file without writing the changes just made by typing the following and pressing the RETURN key:

```
:q!
```

Recovering Lost Text

You might have a serious problem if you delete text and then regret that it was deleted. The editor saves the last nine deleted blocks of text in a set of numbered registers 1 - through 9. (Text consisting of a few words is not saved in these registers.) You can get the *n*th previous deleted block of text back into your file by the command:

```
"np
```

The " indicates that a register name is to follow, *n* is the number of the register you wish to try, and **p** is the **put** command that puts text in the register after the cursor. If this does not bring back the text you wanted, type **u** to undo this and repeat the command using a different numbered register. You can repeat this procedure until you find the correct deleted text.

An easier way to search for the correct register can be to use the dot (.) command to repeat the **put** command. In general the dot (.) command will repeat the last change. As a special case, when the last command refers to a numbered text register, the dot (.) command increments the number of the register before repeating the **put** command. Thus, a sequence of the form

```
"1p
u
.
u
.
and so on
```

will, if repeated long enough, show all the deleted text that was saved. Omit the **u** commands and place all of the text in the numbered registers at one location. Stop after any . command to put just the then-recovered text at one location. The command **P** can also be used rather than **p** to put the recovered text before rather than after the cursor.

Example Nine

1 Access the file containing the text that appears at the end of Example Five:

```
vi filename
```

2 Delete the first line in the first paragraph by typing the following:

```
dd
```

3 Delete the first line in the second paragraph by typing the following:

```
.
```

4 Delete the first line in the third paragraph by typing the following:

```
.
```

5 Delete the first line in the fourth paragraph by typing the following:

```
.
```

6 Move the cursor to any character in the first line of the first paragraph.

7 Put the correct line back ahead of this line by typing the following:

```
"4P
```

The original line is retrieved from register 4.

8 Move the cursor to any character in the first line of the second paragraph.

9 Put the correct line back ahead of this line by typing the following:

```
"3P
```

The original line is retrieved from register 3.

10 Move the cursor to any character in the first line of the third paragraph.

11 Put the correct line back ahead of this line by typing the following:

```
"2P
```

The original line is retrieved from register 2.

12 Move the cursor to any character in the first line of the fourth paragraph.

13 Put the correct line back ahead of this line by typing the following:

```
"1P
```

The original line is retrieved from register 1.

14 Exit your file without writing the changes just made by typing the following and pressing the RETURN key:

```
:q!
```

Copying Another File to the Buffer

While using **vi**, it may be necessary to copy another file into the file you are editing. This can be accomplished using the **:r** command. To copy a file into your file, enter the **:**, a line number that you desire the new text to follow, the **r**, and the name of the file you wish to copy. Then press the RETURN key. The format for this command is

```
:[n]r filename
```

[*n*] can be any line number in your file. If you enter a 0, the copied file will be added before line 1 in your file. If you enter a $, the copied file will be added to the end of your file.

When the file is added, the editor will display at the bottom of the screen the name of the file you copied, the number of lines in that file, and the number of characters it contains. If you do not enter a number in the above command, the file to be copied will be added following the line your cursor was on when you entered the command.

For example, if you wish to write a file called "test" to follow line 10 in your file, enter the following and press the RETURN key:

```
:10r test
```

Example Ten

***Note:** This exercise will require the use of two files. The first one will be the file located at the end of Example Five. The second file should be a copy of any file in your directory.*

1 Access the file contianing text of the file that appears at the end of Example Five by typing the following:

```
$ vi filename
```

2 Move the cursor to any character in the last line of the first paragraph.

3 Copy another file to follow the line the cursor is on by typing the following (remember to press the RETURN key):

```
:r secondfile
```

The second file is copied following the line the cursor is on. The name of the file copied followed by the number of lines and the number of characters appears at the bottom of the screen.

4 Copy the second file to precede this file by typing the following:

```
:0r secondfile
```

5 Copy the second file to follow this file by typing:

```
:$r secondfile
```

6 Exit your file without writing the changes just made by typing the following:

```
:q!
```

7 Access the first file again by typing:

```
$ vi filename
```

8 With the cursor on any character in the first line of the file, add the second file to follow line 29 of this file by typing:

```
:29r secondfile
```

9 Move the cursor to the line immediately preceding the added text.

10 Find the line number by typing the following:

```
:nu
```

The line number and the line you are on appears at the bottom of the screen. You should be on line 29.

11 As indicated at the bottom of the screen, press the RETURN key to continue.

12 Attempt to undo this read by typing:

```
u
```

The wrong lines are deleted.

Anytime you enter a number or the **$** before the **r** and the line number requested is other than line the cursor is on, you cannot undo the read just performed.

13 Exit your file without writing the changes just made by typing the following:

```
:q!
```

Writing Sections of the Buffer to Another File

You can copy all of or a part of your file to another file by using the :**w** command as follows:

```
:[m],[n]w filename
```

The [*m*] represents the line number of the beginning of the text you wish to copy to another file. The [*n*] represents the line number of the last line in the text you wish to copy to another file. A **$** can be entered indicating that you wish to copy to the end of your file. "filename" represents the name of the file you wish to copy the text into. For example, if you wish to write lines 12 through 25 to a new file called "temp" enter the following:

```
:12,25w temp
```

The editor will display at the bottom of the screen the name of the file "temp" you have copied into, the number of lines, and the number of characters entered into the file "temp." If no numbers are entered, the entire file you are in will be copied to the filename entered. If :w (no numbers and no filename) is entered, you rewrite your present buffer contents back to the file you are in.

In many cases, the file you wish to copy information into already exists. You may desire to either add the new information to the end of the other file or to write over the existing information in the other file. Using the example above, write over the file by making the following command entry:

```
:12,25w! temp
```

If you forget the !, the editor will display the following message at the bottom of the screen

```
File exists - "w! temp" to overwrite.
```

If you wish to append to the temp file, make the following command entry:

```
:12,25w temp
```

The editor will display at the bottom of the screen the name of the file "temp" you have copied into, the number of lines, and the number of characters added to the file "temp."

Example Eleven

This example will require the use of two files. The first one will be the file located at the end of Example Five. The second file should be a copy of any file in your directory.

1 Access the file containing the text that appears at the end of Example Five by typing the following:

```
$ vi filename
```

2 Move the cursor to any character in the first line of the second paragraph.

3 Determine the line number the cursor is on by typing the following:

```
:nu
```

The line number is 15. Record this number for later use in a write command.

4 As indicated in the message that appears at the bottom of the screen, press the RETURN key to continue.

The cursor returns to line 15.

5 Move the cursor to any character in the last line of the third paragraph.

6 Determine the line number the cursor is on by typing the following:

```
:nu
```

The line number is 23. The message "Hit return to continue" is displayed at the bottom of the screen. Do not do it.

7 Write the information from lines 15 through 23 into a new file by typing the following:

```
:15,23w newfile
```

The new file name followed by [*Newfile*], *9 lines, 395 characters* is displayed at the bottom of the screen.

8 Write the information from lines 15 through 23 to the end of your second file by typing the following:

```
:15,23wsecondfile
```

The second file name followed by 9 lines, 395 characters is displayed at the bottom of the screen.

9 Exit your file without writing the changes just made by typing the following:

```
:q!
```

Global, Substitutions

When it is required to locate a number of occurrences of text in your file, the **:g (global)** command and the "search for" command can be used with the following formats:

- :[*m*],[*n*]**g**/*text*
- :[*m*],[*n*]**g**/*text*/**p**
- :[*m*],[*n*]**g**/*text*/**nu**

When the first command format is entered, the cursor will move to the last occurrence of the *text* searched for.

When the second command format is entered, the lines containing all occurrences of the *text* searched for are displayed on the screen.

When the third command format is entered, the lines with line numbers containing all occurrences of the *text* searched for are displayed on the screen.

The [*m*] represents the line number where the search will start. The [*n*] represents the line number where the search will stop. If a **$** is entered, the search will continue to the end of your file. No numbers entered (default) will cause all lines in the file to be searched. A **?** substituted for the / will have the same effect in the global search.

Rather than making a global search alone, it may be required to change the text being searched for. This can be accomplished by adding a substitute command to the global search. The following formats can be used for global substitutes:

- :[*m*],[*n*]**g**/*text*/**s**//*newtext*
- :[*m*],[*n*]**g**/*text*/**s**//*newtext*/**p**
- :[*m*],[*n*]**g**/*text*/**s**//*newtext*/**c**

When the first command format is entered, *newtext* will be substituted for *text* at the first occurrence on each line requested in the command. The cursor will be placed at the last occurrence of the changed *newtext.*

When the second command format is entered, *newtext* will be substituted for "text" at the first occurrence on each line requested in the command. The lines containing all occurrences of *newtext* substitutions are displayed on the screen.

When the third command format is entered, you are in a "prompt" mode. The "prompt" mode will allow you to decide to make the change to *newtext* at each occurrence of *text.* The line with the first occurrence of "text" is displayed at the bottom of the screen. Each of the characters in *text* will be replaced by ^ (caret). If you type a **y** followed by pressing the RETURN key, *newtext* will be substituted for *text* in the file. The next line containing *text* will then be displayed with ^'s replacing *text*. If you decided not to make the change to *newtext* for the line displayed, press the RETURN key and the next line with *text* will be displayed. The line displayed may appear as follows:

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

Between / and *text,* a ^ (caret) can be inserted, and only the occurrences of *text* at the beginning of the line will be searched.

Between *text* and /, a **$** can be inserted, and only the occurrences of *text* at the end of the line will be searched.

Example Twelve

1 Access the file that contains the text that appears at the end of Example Five by making the following command entry:

```
$ vi filename
```

2 Find all of the appearances of the word "shell" and print the line numbers by making the following command entry:

```
:g/shell/nu
```

Line numbers 2, 3, 6, 11, 16, 19, and 21 are displayed at the bottom of the screen. "Hit return to continue" is displayed at the bottom of the screen. Do not do it.

3 Change the first four appearances of "shell" to five x's and print these lines by making the following command entry:

```
:2,11g/shell/s//xxxxx/p
```

Change the word "shell" to five z's at the first and last appearances of the word by making the following command entry:

```
        :g/shell/s//zzzzz/c
```

The line with the first appearance of "shell" with ^^^^^ shown underneath "shell" is displayed at the bottom of the screen.

4 Change this word to zzzzz by making the following command entry:

```
y
```

The next line with "shell" is displayed.

5 Do not change this word. Press the RETURN key.

The third line with "shell" is displayed.

6 Do not change this word. Press the RETURN key.

The fourth line with "shell" is displayed

7 Change "shell" to zzzzz.

```
y
```

"Hit return to continue" is displayed at the bottom of the screen.

8 Press the RETURN key.

The cursor is placed at the last change made by the global substitution.

9 Exit your file without writing the changes just made by making the following command entry:

```
:q!
```

Escaping to the Shell

You can get to a shell to execute a single command by giving a **vi** command of the form

```
:!cmd
```

The system will run the single command (**cmd**); and when finished, the editor will ask you to press the RETURN key to continue. When you have finished looking at the output on the screen, you should press the RETURN key, and the editor will clear the screen and redraw it. You can then continue editing. You can also give another : command when it asks you to press the RETURN key. In this case, the screen will not be redrawn.

If you wish to execute more than one command in the shell, then give the command

```
:sh
```

This will give you a new shell. When finished with the shell, end it by pressing CODE-**d**. The editor will clear the screen and continue.

Example Thirteen

This example will require the use of two files. The first one will be the file created at the end of Example Five. The second file should be a copy of any file in your directory.

1 Access the file that contains the text that appears at the end of Example Five by making the following command entry:

```
$ vi filename
```

2 Look at your second file by making the following command entry:

```
:!cat secondfile
```

Your second file is displayed to you.

3 As indicated at the bottom of the screen, press the RETURN key to continue.

The cursor returns to the line you were on before you escaped to the shell.

4 Escape to the shell by making the following command entry:

```
:sh
```

Your system prompt is displayed at the bottom of the screen.

5 Remove the new file you created when you typed in **:15,23w** *newfile* by making the following command entry:

```
rm newfile
```

Your system prompt is returned.

6 Check to see if the file is actually gone by making the following command entry:

```
ls
```

A listing of the files in your present directory appears. The new file just removed should not be in the listing. The system prompt is displayed after this listing.

7 Return to the file you were editing by pressing CODE-**d**.

The cursor returns to the place it was located before you escaped to the shell.

8 Exit your file without writing the changes just made by making the following command entry:

```
:q!
```

Abbreviated Example

This example is an abbreviated example for those who require a limited use of the editor. Refer to the other exercises for details. Type what is shown in the displays. Use the BACKSPACE key to type over errors.

1 Enter the editor to edit a file named "abbex" by making the following command entry and pressing the RETURN key:

```
$ vi abbex
```

Wait for screen display.

2 Enter the append mode.

```
a
```

3 Append the following information (press the RETURN key after completing each line):

```
When youall have successfully logged in, a program
called the shell is not listening to your terminal.
The shell reads the line we type, splits it into
a command name and its arguments, and the
command. A simply an executable program.
located in a file. Narmally, the shell looks first
in your current directory for a program with your
directories. This text is wrong
system-provided commands except that they are kept
for the shell to find them there.
```

4 Leave the append mode by pressing the GO key.

5 Move the cursor to the fourth line from the top of the file.

```
k
```

Repeat until your cursor is on the correct line. If you overrun, use the **j** key to backup.

6 Move your cursor to the space following the second "and."

```
l
```

Repeat until you arrive at the correct spot. If you overrun, use the **h** key to backup.

7 Enter the append mode.

```
a
```

8 Enter the following and press the GO key:

```
executes
```

9 Move the cursor to the next line by pressing the RETURN key.

10 Using the SPACEBAR move the cursor to the s in simply.

Repeat until you reach the correct spot. If you overrun, use the BACKSPACE key.

11 Enter the insert mode.

```
i
```

12 Enter the following and press the GO key:

```
command is
```

13 Move the cursor down two more lines by pressing the RETURN key twice.

14 Open a line below the line the cursor is on.

```
o
```

15 Enter the following and press the GO key:

```
command name, and if none is there, then in system
```

16 Move the cursor to any character in the last line in your file.

```
G
```

17 Open a line above the line the cursor is on.

```
O
```

18 Enter the following text and press the GO key:

```
in directories where the shell can find them.
You can  (Press the RETURN key)
 also keep commands in your own directories
and arrange  (Press the GO
key)
```

19 Move the cursor to the beginning of the file.

```
1G
```

Move the cursor to the a in "youall" by moving a word and three spaces.

```
wlll
```

20 Remove "all."

```
xxx
```

21 Move the cursor to "not" in the second sentence.

```
4w
```

22 Remove "not."

```
dw
```

23 Delete this line.

```
dd
```

24 Restore the line.

```
u
```

25 Write the information you have entered thus far but stay in the editor by making the following command entry and pressing the RETURN key.

```
:w
```

26 Move the cursor to "we" in the next line.

```
5w
```

27 Change "we" to "you." (Press GO.)

```
cwyou
```

Move the cursor to "Narmally" by doing a "search for." (Press RETURN.)

```
/N
```

28 Replace the next "a" with an "o."

```
lro
```

29 Press the RETURN key three time to move the cursor down three lines. Then use the find **f** character command to find "This."

```
fT
```

30 Replace the text to the end of the line. (Press the GO key.)

```
c$There is nothing special about
```

31 Move the cursor to the beginning of the screen display.

```
H
```

32 Change this line and the next line to a bunch of z's. Press the GO key.)

```
2cczzzzzzzzzzzzzzzzzzzzz
```

33 Restore the original lines.

```
u
```

34 Scroll down the file.

```
^d
```

35 Scroll up.

```
^u
```

36 Move to the last line on the screen.

```
L
```

37 Move the cursor to the end of the line.

```
$
```

38 Move the cursor to the beginning of the line.

```
0
```

39 Write the first five lines in this file to another file. (Press the RETURN key.)

```
:1,5w abbexn
```

40 Add another known file to the this file following the line the cursor is on. (Press the RETURN key.)

:r *yourfilename*

41 Remove the added information.

```
u
```

42 The file you have just completed should look like the file at the end of Example Three.

43 Write and quit.

```
zz
```

At the bottom of the screen appears your file name in quotes followed by [*Newfile*], the number of lines, and the number of characters.

Visual Editor (vi)—Advanced

The reader should be familiar with the information covered in Section Eleven before using the information in this section.

Getting Started

Specifying Terminal Type

Before you can start **vi**, the CENTIX system requires you to identify the kind of terminal you are using. If your terminal does not have a code, consult with one of the staff members on your system to determine the code for your terminal. A code can be assigned and a description for the terminal created. Assuming you have a PT 1500 terminal, use the following command sequence

```
$ TERM=pt
$ export TERM
```

to identify your terminal.

If you want your terminal type established automatically when you log in, place the above commands in your .profile file.

Editing a File

After telling the system the kind of terminal you have, make a copy of a file you are familiar with and run **vi** on this file with the command

```
$ vi filename
```

Replace *filename* with the name of the copy file just created. The screen should clear and the text of your file appear on the screen. If something else happens, perhaps you gave the system an incorrect terminal type code, in which case the editor may have just made a mess out of your screen. This happens when **vi** sends control codes for one kind of terminal to some other kind of terminal. In this case, type the keys **:q** (colon and the q key) and then press the RETURN key. This should get you back to the command level interpreter. Figure out what you did wrong (ask someone else if necessary) and try again.

Another thing that can go wrong is you typed the wrong file name and the editor just printed an error diagnostic. In this case, follow the above procedure for getting out of the editor; then try again, spelling the file name correctly.

If the editor does not seem to respond to the commands you enter, try sending an interrupt to it by pressing the DELETE key, and then typing the **:q** command, followed by pressing the RETURN key.

Editor Copy in Buffer

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

Arrow Keys

The editor command set is independent of the terminal you are using. On most terminals with cursor positioning keys, these keys will also work within the editor. If you do not have cursor positioning keys or even if you do, you can use the **h**, **j**, **k**, and **l** keys as cursor positioning keys.

- **h** moves cursor to the left (like CODE-**h** which is a backspace).
- **j** moves cursor down (in the same column).
- **k** moves cursor up (in the same column).
- **l** moves cursor to the right.

Leaving the Editor

After you have worked with the editor for a while and wish to do something else, give the **ZZ** command to leave the editor. This writes the contents of the editor buffer (including any changes made) back into the file you are editing and then quits the editor.

An editor session can also be terminated with the **:q!** command All commands read from the last display line can also be terminated by pressing the GO and the RETURN keys. This is a dangerous but occasionally essential command that ends the editor session and discards all your changes. You need to know about this command in case you change the editor copy of a file you wish only to look at. Be very careful not to give this command when you really want to save the changes you have made.

Moving Around in the File

Scrolling and Paging

The editor has a number of commands for moving around in the file. The most useful of these is generated by pressing the CODE and **d** keys at the same time, or a CODE-**d**. We will use this two-character notation for referring to these CODE keys from now on.

As you know now, if you tried typing CODE-**d**, this command scrolls down in the file. The **d** thus stands for down. Many editor commands are mnemonic, and this makes them much easier to remember. For instance, the command to scroll up is CODE-**u**. Many dumb terminals cannot scroll up at all, in which case pressing CODE-**u** clears the screen and refreshes it with a line farther back in the file at the top.

If you want to see more of the file below where you are, press CODE-**e** to expose one more line at the bottom of the screen leaving the cursor where it is. The CODE-**y** (which is hopelessly nonmnemonic, but next to CODE-**u** on the keyboard) command exposes one more line at the top of the screen.

There are other ways to move around in the file; the keys CODE-**f** and CODE-**b** move forward and backward a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file using these commands rather than the CODE-**d** and CODE-**u**.

There is a difference between scrolling and paging. If you are trying to read the text in a file, pressing CODE-**f** to move forward a page gives you only a little context at which to look back. Scrolling (CODE-**d**) on the other hand gives more context and functions more smoothly. You can continue to read the text while scrolling is taking place.

Searching, Goto, and Previous Context

Another way to position yourself in the file is by giving the editor a string to search for. Type the / character followed by a string of characters, and press the RETURN key. The editor will position the cursor at the next occurrence of this string. Then try typing **n** to go on to the next occurrence of this string. The **?** character will search backward from where you are and is otherwise like /.

These searches will normally wraparound the end of the file and find the string even if it is not on a line in the direction you search (provided the string is in the file). You can disable this wraparound in scans by giving the command:

```
:se nowrapscan
```

or

```
:se nows
```

If the search string you give the editor is not present in the file, the editor will print a diagnostic on the last line of the screen; and the cursor will return to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with an caret symbol (^). To match only at the end of a line, end the search string with a **$**. Thus

```
/^first
```

will search for the word "first" at the beginning of a line, and

```
/last$
```

searches for the word "last" at the end of a line.

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

```
    :se nomagic
```

command in EXINIT in your environment.

The command **G** when preceded by a number will position the cursor at that line in the file. Thus **1G** will move the cursor to the first line of the file. If you give **G** no count, then the cursor moves to the end of the file.

If you are near the end of the file and the last line is not at the bottom of the screen, the editor will place only the tilde character (~) on each remaining line. This indicates that the last line in the file is on the screen; that is, the ~ lines are beyond the end of the file.

Determine the state of the file you are editing by typing a CODE-**g** command. The editor will show the name of the file you are editing, the number of the current line, the number of lines in the buffer, and the percentage of the way through the buffer the cursor is located. Try doing this now and remember the number of the line you are on. Give a **G** command to get to the end and then another **G** command to get back where you started.

You can also get back to a previous position by using the command \`\` (two grave accents). This is often more convenient than **G** because it requires no advance preparation. Try giving a **G** or a search with / or **?** and then a \`\` to get back where you started. If you accidentally type **n** or any command that moves you far away from a context of interest, you can quickly get back by typing \`\`.

Moving Around on the Screen

Now try just moving the cursor around on the screen. If your terminal has arrow keys (four or five keys with arrows going in each direction) try them and convince yourself that they work. If the arrow keys do not work, you can always use **h**, **j**, **k**, and **l**. Experienced users of **vi** prefer these keys to arrow keys because they are usually right beneath their fingers.

Type the + key. Each time you do notice that the cursor advances to the next line in the file at the first nonwhite position on the line. The - key is like + but goes the other way. These are very common keys for moving the cursor up and down lines in the file.

***Note:** If the cursor goes off the bottom or top of the screen when using these keys, then the text will scroll down (and up, if possible) to bring a line at a time into view. The key has the same effect as the + key.*

The **vi** editor also has commands to take you to the top, middle, and bottom of the screen. The **H** command will take you to the top line (home) on the screen. Try preceding it with a number as in **3H**. This will take you to the third line on the screen. Many **vi** commands take preceding numbers and do interesting things with them. Try **M**, which takes you to the middle line on the screen, and **L**, which takes you to the last line on the screen. The **L** command also takes counts, the **5L** command will take you to the fifth line from the bottom. The **M** command does not take counts.

There are two control characters that move the cursor up or down a line, but keep it in the same column. CODE-**n** causes the cursor to move to the same column of the next line. To move to the same column of the previous line, use the CODE-**p** command.

Moving Within a Line

Now try picking a word on some line on the screen, not the first word on the line. Move the cursor (using the RETURN key and -) to be on the line where the word is. Try typing the **w** key. This will advance the cursor to the next word on the line. Try typing the **b** key to back up words in the line. Also try the **e** key which advances you to the end of the current word rather than to the beginning of the next word. Also try the SPACEBAR, which moves the cursor right one character, and the BACKSPACE key (or CODE-**h**), which moves left one character. The **h** key works as CODE-**h** does and is useful if you do not have a BACKSPACE key. Also, as noted above, the **l** key will move the cursor to the right.

If the line has punctuation, you may have noticed that the **w** and **b** keys stop at each group of punctuation. You can also go backward and forward without stopping at punctuation by using **W** and **B** rather than the lowercase equivalents. Think of these as bigger words. Try these on a few lines with punctuation to see how they differ from the lowercase **w** and **b**.

The word keys wraparound the end of line rather than stopping at the end. Try moving to a word on a line below where you are by repeatedly typing **w**.

The View Editor

If you want to use the editor to look at a file rather than to make changes, invoke it as view instead of **vi**. This will set the readonly option that will prevent you from accidently overwriting the file.

Making Simple Changes

Inserting

One of the most useful commands is the **insert** (**i**) command. After you type i, everything you type until you press the GO key is inserted into the file. Try this now; position yourself on some word in the file and try inserting text before this word. If you are on a dumb terminal, it will seem that some of the characters in your line have been overwritten, but they will reappear when you press the GO key.

Now try finding a word that can, but does not, end in an "s". Position yourself at that word and type **e** (move to end of word), **a** (append), **s** and press the GO key (terminate the textual insert). This sequence of commands can be used to pluralize a word.

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

- **i** places text to the left of the cursor.
- **a** places text to the right of the cursor.

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lowercase key and the other is given by an uppercase key. In these cases, the uppercase key often differs from the lowercase key in its sense of direction, with the uppercase key working backward and/or up, while the lowercase key moves forward and/or down.

It is often the case that you want to add new lines to the file you are editing before or after some specific line in the file. Find a line where this makes sense and then give the **o** command to create a new line after the line you are on or the **O** command to create a new line before the line you are on. After you create a new line in this way, the text you type until you press the GO key is inserted on the new line.

Whenever you are typing in text, you can give many lines of input or just a few characters. To type in more than one line of text, press **a** at the middle of your input. A new line will be created for text, and you can continue to type. If you are on a slow and dumb terminal, the editor may choose to wait to redraw the tail of the screen and will let you type over the existing screen lines. This avoids the lengthy delay that would occur if the editor attempted to keep the tail of the screen always up to date. The tail of the screen will be fixed, and the missing lines will reappear when you press the GO key.

While inserting new text, you can use the characters normally used at the system command level (usually CODE-**h** or #) to backspace over the last character typed. The character used to kill input lines (usually @, CODE-**x**, or CODE-**u**) can be used to erase the input you have typed on the current line. In fact, the CODE-**h** (backspace) always works to erase the last input character, regardless of what your erase character is. The CODE-**w** command will erase a whole word and leave you after the space following the previous word. It is useful for quickly backing up in an insert. The following conditions should be noted:

- When you backspace during an insertion, the characters you backspace over are not erased; the cursor moves backward, and the characters remain on the display. This is often useful if you are planning to type in something similar. In any case, the characters disappear when you press the GO key. If you want to get rid of them immediately, press the GO key and then **a** again.
- You cannot erase characters that you did not insert, and you cannot backspace around the end of a line. If you need to back up to the previous line to make a correction, just press the GO key and move the cursor back to the previous line. After making the correction, return to where you were and use the **insert** or **append** command again.

Making Small Corrections

You can make small corrections in existing text quite easily. Find a single character that is wrong or just pick any character. Use the arrow keys to find the character, or get near the character with the word motion keys and then either backspace (press the BACKSPACE key, CODE-**h**, or just **h**) or space (using the SPACEBAR) until the cursor is on the wrong character. If the character is not needed, then type the **x** key; this deletes the character from the file. It is analogous to the way you x out characters when you make mistakes on a typewriter.

If the character is incorrect, replace it with the correct character by giving the **r***c* command, where *c* is the correct character. If the character that is incorrect should be replaced by more than one character, enter the following command and press the GO key:

```
s string
```

which substitutes a string of characters. If there are a small number of characters that are wrong, precede **s** with a count of the number of characters to be replaced. Counts are also useful with **x** to specify the number of characters to be deleted.

Making Corrections with Operators

You already know almost enough to make changes at a higher level. All you need to know now is that the **d** key acts as a delete operator and the **c** key acts as a change operator.

- The **dw** command deletes a following word.
- The **db** command deletes a preceding word.
- The **d**-SPACEBAR command deletes a single character and is equivalent to the **x** command.
- The . command repeats the last command that made a change.- The **cw** command changes the text of a single word. It is followed with replacement text after pressing the GO key.

Note: *The end of the text to be changed is marked with the $ character so that you can see this mark as you are typing in new text material.*

Operating on Lines

It is often the case that you want to operate on lines. Find a line you want to delete and type dd, the d operator twice. This will delete the line. If you are on a dumb terminal, the editor may erase the line on the screen replacing it with a line with only an @ on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen that would be necessary to close up the hole created by the deletion on a terminal without a delete line capability.

Repeating the **c** operator twice (**cc**) will change a whole line erasing its previous contents and replacing them with text you type before pressing the GO key. The **S** command is a convenient synonym for **cc**, by analogy with **s**. Think of **S** as a substitute on lines, while **s** is a substitute on characters.

You can delete or change more than one line by preceding the **dd** or **cc** with a count (**5dd** deletes 5 lines). You can also give a command like **dL** to delete all the lines up to and including the last line on the screen or **d3L** to delete through the third from the bottom line. One subtle point here involves using the / (search) after a **d**. This will normally delete characters from the current position to the point of the match. If you desire to delete whole lines including the two points, give the pattern as /**pat**/+**0**, a line address.

Note: *The **vi** editor lets you know when you change a large number of lines so that you can see the extent of the change. It will also always tell you when a change you make affects text that you cannot see.*

Undoing

Suppose the last change made was incorrect; use the **insert**, **delete**, and **append** commands to put the correct material back. However, since it is often the case that we regret a change or make a change incorrectly, the editor provides an **undo** (**u**) command to reverse the last change made. A **u** also undoes a **u**.

The **u** command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the original state of the line back. The **U** command restores the current line to the state before you started changing it.

You can recover text that you deleted even if the **u** command will not bring it back. This recovery procedure is covered in this section in the subsection "Recovering Lost Lines."

Moving About, Rearranging, and Duplicating Text

Low Level Character Motions

Move the cursor to a line where there is a punctuation or a bracketing character such as a parenthesis or a comma or period. Try the command **f***x* where *x* is the character sought.

This command finds the next *x* character to the right of the cursor in the current line. Try then typing **a** ;, which finds the next instance of the same character. By using the **f** command and then a sequence of ;'s, you can often get to a particular place in a line much faster than with a sequence of word motions or pressing the SPACEBAR. There is also an **F** command, that is like **f**, but searches backward. The ; command repeats **F** also.

When operating on the text in a line, it is often desirable to delete characters including the first instance of a character. Try **df***x* for some *x* and notice that the *x* character is deleted. Undo this with u and then try **dt***x* (the t stands for "to") to delete up to the next *x*, but not the *x*. The command **T** is the reverse of t.

When working with the text of a single line, an ^ moves the cursor to the first nonwhite position on the line, and a **$** moves it to the end of the line. Thus, **$a** will append new text at the end of the current line.

Your file may have tab characters (^**i**) in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every eight positions by default. Tab stops can be set by a command of the form

```
:se ts= x
```

where *x* is four to set tab stops every four columns. The tab stop setting has an effect on screen representation within the editor. When the cursor is at a tab, it sits on the last of the several spaces that represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may have nonprinting characters in it. On the screen, nonprinting characters resemble a ^ character adjacent to another. Spacing or backspacing over the character reveals that the two characters are, like the spaces representing a tab character, a single character.

The editor sometimes discards control characters, depending on the character and the setting of the *beautify* option if you attempt to insert them in your file. You can get a control character in the file by beginning an insert and then typing CODE-**v** before the control character. The CODE-**v** quotes the following character causing it to be inserted directly into the file.

Higher Level Text Objects

In working with a document, it is often advantageous to work in terms of sentences, paragraphs, and sections.

- The (and) operations move to the beginning of the previous and next sentence, respectively. Thus the **d)** command will delete the rest of the current sentence. The **d(** command will:

  delete the previous sentence if you are at the beginning of the current sentence

  or

  delete the current sentence up to your present position if you are not at the beginning of the current sentence.

- A sentence is defined to end at a ., !, or ? followed by either the end of a line or two spaces. Any number of),], ", and ' closing characters may appear after the ., !, or ? and before the spaces or end of line.
- The { and } operations move over paragraphs.
- A paragraph begins after each empty line and also at each of a set of paragraph macros specified by the pairs of characters in the definition of the string-valued option paragraphs. The default setting for this option defines the paragraph macros of the -ms and -mm macro packages, such as the .IP, .LP, .PP, and .QP, .P and .LI macros.

  You can easily change or extend this set of macros by assigning a different string to the paragraphs option in your EXINIT. The **.bp** request is also considered to start a paragraph. Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs.

- The [[and]] operations move over sections. They require the operation character to be doubled because they can move the cursor from where it currently is. While it is easy to get back with the `` command, these commands would still be frustrating if they were easy to hit accidentally.
- Sections in the editor begin after each macro in the sections option (normally, .NH, .SH, .H, and .HU) and each line with a formfeed CODE-l in the first column. Section boundaries are always line and paragraph boundaries.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking at it using the section commands. Section commands interpret a preceding count as a different window size in which to redraw the screen at the new location, and this window size is the base size for newly drawn windows until another size is specified. This is very useful if you are on a slow terminal and are looking for a particular section. You can give the first section command and a small count to see each successive section heading in a small window.

Rearranging and Duplicating Text

The editor has a single unnamed buffer where the last deleted or changed text is saved, and a set of named buffers a-z that you can use to save copies of text and to move text around in your file and between files.

The **y** operator yanks a copy of the object that follows into the unnamed buffer. If preceded by a buffer name, "xy, where *x* is replaced by a letter a-z, it places the text in the named buffer. The text can then be put back in the file with the commands **p** and **P**; the **p** command puts the text after or below the cursor, while **P** puts the text before or above the cursor.

If the text you yank forms a part of a line or is an object such as a sentence that partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if you use **P**). If the yanked text forms whole lines, they will be put back as whole lines without changing the current line. In this case, the put acts much like an **o** or **O** command.

Try the **YP** command. This makes a copy of the current line and leaves you on this copy, which is placed before the current line. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** will also make a copy of the current line and place it after the current line. You can give **Y** a count of lines to yank, and thus duplicate several lines; try **3YP**.

To move text within the buffer, delete it in one place and put it back in another. Precede a delete operation by the name of a buffer in which the text is to be stored as in **"a5dd** deleting five lines into the named buffer a. Then move the cursor to the eventual resting place of these lines and do an **"ap** or **"aP** to put them back. In fact, you can switch and edit another file before you put the lines back by giving a command of the form

```
:e name
```

where *name* is the name of the other file you want to edit. You will have to write back the contents of the current editor buffer (or discard them) if you have made changes before the editor will let you switch to the other file. An ordinary **delete** command saves the text in the unnamed buffer so that an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you change files; so to move text from one file to another, use a named buffer.

High-Level Commands

Writing, Quitting, and Editing New Files

So far we have seen how to enter the **vi** editor and to write out our file using either **ZZ** or **:w** commands. The first command exits from the editor (writing if changes were made); the second command writes and stays in the editor.

If you have changed the editor copy of the file but do not wish to save your changes either because you messed up the file or decided that the changes are not an improvement, then you can give the command

```
:q!
```

to quit from the editor without writing the changes. You can also reedit the same file (start over) by giving the command

```
:e!
```

These commands should be used only rarely and with caution since it is not possible to recover the changes you have made after you discard them in this manner.

You can edit a different file without leaving the editor by giving the command

```
:e filename
```

If you have not written out your file before you try to do this, then the editor will tell you this and delay editing the other file. You can then give the command

```
:w
```

(to save your changes) and then the

```
:e filename
```

command again or carefully give the command

```
:e! filename
```

which edits the other file discarding the changes you have made to the current file. To have the editor automatically save changes, include *set autowrite* in your EXINIT and use **:n** instead of **:e**.

Escaping to a Shell

You can get to a shell to execute a single command by giving a **vi** command of the form

```
:!command
```

The system will run the single *command*; and when finished, the editor will ask you to press the RETURN key to continue. When you have finished looking at the output on the screen, press the RETURN key and the editor will clear the screen and redraw it. Then continue editing. You can also give another : command when it asks you the press the RETURN key. In this case, the screen will not be redrawn.

If you wish to execute more than one command in the shell, give the command

```
    :sh
```

This will give you a new shell. When finished with the shell, end it by pressing CODE-**d**. The editor will clear the screen and continue.

Marking and Returning

The command `` returns you to the previous place after a motion of the cursor by a command such as /, **?**, or **G**. You can also mark lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command **m***x*, where you should pick some letter for *x*, such as an "a." Then move the cursor to a different line (any way you like) and type `a. The cursor will return to the place you marked. Marks last only until you edit another file.

When using operators such as **d** and referring to marked lines, it is often desirable to delete whole lines rather than deleting to the exact position in the line marked by **m**. In this case, use the form '*x* rather than `*x*. Used without an operator, '*x* will move to the first nonwhite character of the marked line; similarly, '' moves to the first nonwhite character of the line containing the previous context mark ``.

Adjusting the Screen

If the screen image is messed up because of a transmission error to your terminal or because some program other than the editor wrote output to your terminal, type a CODE-**l**, the ASCII form-feed character, to cause the screen to be refreshed.

On a dumb terminal if there are @ lines in the middle of the screen as a result of line deletion, get rid of these lines by typing CODE-**r** to cause the editor to retype the screen closing up these holes.

If you wish to place a certain line on the screen at the top, middle, or bottom of the screen, position the cursor to that line and then give a **z** command. After entering the **z** command, press the RETURN key if you want the line to appear at the top of the window, **a** . if you want it at the center, or **a** - if you want it at the bottom.

Special Topics

Editing on Slow Terminals

When you are on a slow terminal, it is important to limit the amount of output generated to your screen so that you will not suffer long delays waiting for the screen to be refreshed. We have already pointed out how the editor optimizes the updating of the screen during insertions on dumb terminals to limit the delays and how the editor erases lines to @ when they are deleted on dumb terminals.

The use of the slow terminal insertion mode is controlled by the slowopen option. You can force the editor to use this mode even on faster terminals by giving the command

```
:se slow
```

If your system is sluggish, this helps lessen the amount of output coming to your terminal. Disable this option with the command

```
:se noslow
```

The editor can simulate an intelligent terminal on a dumb one. Try giving the command

```
:se redraw
```

This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. Disable this with the command

```
:se noredraw
```

The editor also makes editing more pleasant at low speed by starting editing in a small window and letting the window expand as you edit. This works particularly well on intelligent terminals. The editor can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try the editor on an intelligent terminal to see how this works.

Control the size of the window that is redrawn each time the screen is cleared by giving window sizes as argument to the commands that cause large screen motions:

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

Thus if you are searching for a particular instance of common string in a file, precede the first search command by a small number, such as 3; and the editor will draw three line windows around each instance of the string it locates.

You can easily expand or contract the window and place the current line as you choose by giving a number with the **z** command (after the **z** and before pressing the RETURN key, ., or -). Thus the command **z5.** redraws the screen with the current line in the center of a five line window.

Note: *The command* ***5z.*** *has an entirely different effect placing line 5 in the center of a new window.*

If the editor is redrawing or otherwise updating large portions of the display, interrupt this updating by pressing the DELETE key. If you do this, you may partially confuse the editor about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by pressing CODE-**l**; or move or search again, ignoring the current state of the display.

Options, Set, and Editor Startup Files

The editor has a set of options, some have been mentioned above. They are as follows:

autoindent, ai default: noai

Can be used to ease the preparation of structured.program text. At the beginning of each **append**, **change**, or **insert** command or when a new line is opened or created by an append, change, insert, or substitute operation, **vi** looks at the line being appended after, the first line changed, or the line inserted before and calculates the amount of white space at the start of the line. It then aligns the cursor at the level of indentation so determined.

If the user then types in lines of text, the lines will continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line will start aligned with the first nonwhite character of the previous line. To back the cursor to the preceding tabstop, press CODE-**d**. The tabstops (going backwards) are defined as multiples of the shiftwidth option. You cannot backspace over the indent except by sending an end-of-file with a CODE-**d**.

Specially processed in this mode is a line with no character added to it, that turns into a completely blank line (the white space provided for the autoindent is discarded). Also specially processed in this mode are lines beginning with an ^ and immediately followed by a CODE-**d**. This causes the input to be repositioned at the beginning of the line while retaining the previous indent for the next line. Similarly, a **0** followed by a CODE-**d** repositions at the beginning without retaining the previous indent.

The autoindent option does not happen in **global** commands or when the input is not a terminal.

autoprint,ap default: ap

Causes the current line to be printed after each **delete**, **copy**, **join**, **move**, **substitute**, **t**, **undo**, or **shift** command. This has the same effect as supplying a trailing **p** to each such command. The autoprint is suppressed in globals, and only applies to the last of many commands on a line.

autowrite,aw default: noaw

Causes the contents of the buffer to be written to the current file if you have modified it and gives a **next**, **rewind**, **tab**, or ! command, or a ^^ (switch files) or ^] (tag goto) command in visual.

***Note:** The command does not* autowrite. *In each case, there is an equivalent way of switching when the autowrite option is set to avoid the autowrite (**ex** for **next, rewind!** for **rewind, tag!** for **tag, shell** for !, and :e # and a :ta! command from within visual).*

beautify, bf default: nobeautify

Causes all control characters except tab, newline, and formfeed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. The beautify option does not apply to command input.

directory, dir default: dir=/tmp

Specifies the directory in which **ex** places its buffer file. If this directory in not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edcompatible default: noedcompatible

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

errorbells,eb default: noeb

Error messages are preceded by a bell. Bell ringing in open and visual mode on errors is not suppressed by setting noeb. If possible, the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht default: ht=8

Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

ignorecase,ic default: noic

All uppercase characters in the text are mapped to lowercase in regular expression matching. In addition, all uppercase characters in regular expressions are mapped to lowercase except in character class specifications.

lisp default: nolisp

The autoindent option indents appropriately for lisp code, and the **()**, { }, [[, and]] commands in open and visual modes are modified to have meaning for lisp.

list default: nolist

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

magic default: magic for **ex** and **vi**

If nomagic is set, the number of regular expression metacharacters is greatly reduced with only ^ and **$** having special effects. In addition, the metacharacters ~ and **&** of the replacement pattern are treated as normal characters. All the normal metacharacters may be made magic when nomagic is set by preceding them with a \.

mesg default: mesg

Causes write permission to be turned off to the terminal while in visual mode, if nomesg is set.

number,nu default: nonumber

Causes all output lines to be printed with line numbers. In addition, each input line will be prompted for by supplying the line number it will have.

open default: open

If noopen, the commands open and visual are not permitted.

optimize, opt default: optimize

Throughput of text is expedited by setting the terminal not to do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

paragraphs, para default: para=IPLPPPQPP LIbp

Specifies the paragraphs for the { and } operations in open and visual mode. The pairs of characters in the option's value are the names of the macros that start paragraphs.

prompt default: prompt

Command mode input is prompted for with a colon (:).

redraw default: noredraw

The editor simulates (using great amounts of output) an intelligent terminal on a dumb terminal (for example, during insertions in visual, the characters to the right of the cursor position are refreshed as each input character is typed). This option is useful only at very high speed.

remap default: remap

If on, macros are repeatedly tried until they are unchanged. For example, if o is mapped to O, and O is mapped to I, then if remap is set, o will map to I, but if noremap is set, it will map to O.

report default: report=5

Specifies a threshold for feedback from commands. Any command that modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as **global, open, undo,** and **visual,** that have potentially more far-reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus, notification is suppressed during a **global** command on the individual commands performed.

scroll default: scroll=1/2 window

Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode and the number of lines printed by a command mode **z** command (double the value of scroll).

sections default: sections=SHNHH HU

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

shell, sh default: sh=/bin/sh

Gives the pathname of the shell forked for the shell escape command ! and by the **shell** command. The default is taken from SHELL in the environment if present.

shiftwidth, sw default: sw=8

Gives the width a software tabstop used in reverse tabbing with CODE-**d** when using autoindent to append text and by the shift commands.

showmatch, sm default: nosm

In open and visual mode when a) or } is typed, it moves the cursor to the matching (or { for one second if this matching character is on the screen. Extremely useful with lisp.

slowopen, slow terminal dependent

Affects the display algorithm used in visual mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent.

tabstop, ts default: ts=8

The editor expands tabs in the input file to be on tabstop boundaries for the purposes of display.

taglength, tl default: tl=0

Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

tags default: tags=tags/usr/lib/tags

A path of files to be used as tag files for the **tag** command. A requested tag is searched for in the specified files, sequentially. By default, files called tags are searched for in the current directory and in /usr/lib (a master file for the entire system).

term from environment TERM

The terminal type of the output device.

terse default: noterse

Shorter error diagnostics are produced for the experienced user.

warn default: warn

Warn if there has been "[No write since last change]" before a ! command escape.

window default: window=speed dependent

The number of lines in a text window in the **visual** command. The default is eight at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

w300, w1200, w9600

These are not true options but set window only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.

wrapscan, ws default: ws

Searches that use regular expressions in addressing will wraparound past the end of the file.

wrapmargin, wm default: wm=0

Defines a margin for automatic wrap over of text during input in open and visual modes.

writeany, wa default: nowa

Inhibit checks normally made before **write** commands allowing a write to any file that the system protection mechanism will allow.

The options are of three kinds: numeric, string, and toggle. Set numeric and string options by a statement of the form

set opt=val

and toggle options can be set or not set by statements of one of the forms

***set** opt*

set no*opt*

These statements can be placed in your EXINIT in your environment or given while you are running **vi** by preceding them with a : and pressing RETURN key.

You can get a list of all options that you have changed with the command

```
:set
```

or the value of a single option by the command

```
:set opt?
```

A list of all possible options and their values is generated by

```
:set all
```

Set can be abbreviated **se**. Multiple options can be placed on one line, for example:

```
:se ai aw nu
```

Options set by the **set** command last only while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be accomplished by creating a list of **ex** commands that are to be run every time you start **ex**, **edit**, or **vi** (all commands that start with : are **ex** commands). A typical list includes a set command and possibly a few map commands (on V3 editors). Since it is advisable to get these commands on one line, they can be separated with the |character; for example:

```
set ai aw terse|map @ dd|map # x
```

which establishes the **set** command options autoindent, autowrite, terse, makes @ delete a line (the first map), and makes # delete a character (the second map). This string should be placed in the variable EXINIT in your environment. Using the shell, put these lines in the file .profile in your home or working directory

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

or put the line in the file exrc in your home directory.

```
set ai aw terse|map @ dd|map # x
```

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

Recovering Lost Lines

You might have a serious problem if you delete a number of lines and then regret that they were deleted. The editor saves the last nine deleted blocks of text in a set of numbered registers 1 through 9.

You can get the *n*th previous deleted text back in your file by the command "*n***p**. The " here says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and **p** is the **put** command that puts text in the buffer after the cursor. If this does not bring back the text you wanted, type **u** to undo this and then . (period) to repeat the **put** command.

In general, the . command will repeat the last change. As a special case, when the last command refers to a numbered text buffer, the . command increments the number of the buffer before repeating the command. Thus a sequence of the form

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

will, if repeated long enough, show you all the deleted text that has been saved for you. Omit the **u** commands here to gather up all this text in the buffer or stop after any . command to keep just the then-recovered text. The command **P** can also be used rather than **p** to put the recovered text before rather than after the cursor.

Recovering Lost Files

If the system crashes, you can recover the work you were doing to within a few changes. You will normally receive mail when you next login giving you the name of the file that has been saved for you. You should then change to the directory where you were when the system crashed and give a command of the form

```
$ vi -r filename
```

replacing name with the name of the file you were editing. This will recover your work to a point near where you left off.

Note: *In rare cases, some of the lines of the file may be lost. The editor will give you the numbers of these lines and the text of the lines will be replaced by the string LOST. These lines will almost always be among the last few that you changed. You can either choose to discard the changes that you made (if they are easy to remake) or to replace the few lost lines by hand.*

You can get a listing of the files that are saved for you by giving the command

```
$ vi -r
```

If there is more than one instance of a particular file saved, the editor gives the newest instance each time you recover it. You can thus get an older saved copy back by first recovering the newer copies.

For this feature to work, **vi** must be correctly installed by a superuser on your system, and the mail program must exist to receive mail. The invocation

```
$ vi -r
```

will not always list all saved files, but they can be recovered even if they are not listed.

Continuous Text Input

When typing in large amounts of text, it is convenient to have lines broken near the right margin automatically. You can cause this to happen by giving the command

```
:se wm=10
```

This causes all lines to be broken at a space at least ten columns from the right-hand edge of the screen.

If the editor breaks an input line and you wish to put it back together, tell it to join the lines with **J**. Give **J** a count of the number of lines to be joined (**3J** to join 3 lines). The editor supplies white space, if appropriate, at the juncture of the joined lines and leaves the cursor at this white space. Kill the white space with x if you do not want it.

Features for Editing Programs

The editor has a number of commands for editing programs. The thing that most distinguishes editing of programs from editing of text is the desirability of maintaining an indented structure to the body of the program. The editor has an autoindent facility to help generate correctly indented programs.

To enable this facility, give the command

```
:se ai
```

Now try opening a new line with **o** and then type some characters on the line after a few tabs. If you now start another line, notice that the editor supplies white space at the beginning of the line to line it up with the previous line. You cannot backspace over this indentation, but you can use CODE-**d** to backtab over the supplied indentation.

Each time you press CODE-**d** you back up one position, normally to an eight-column boundary. This amount is settable; the editor has an option called shiftwidth that you can set to change this value. Try giving the command

```
:see sw=4
```

and then experimenting with autoindent again.

For shifting lines in the program left and right, there are operators < and >. These shift the lines you specify right or left by one shiftwidth. Try < < and > > to shift one line left or right, and <**L** and >**L** to shift the rest of the display left and right.

If you have a complicated expression and wish to see how the parentheses match, put the cursor at a left or right parenthesis and type **%**. This will show you the matching parenthesis. This works also for braces ({ }) and brackets ([ ]).

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

Filtering Portions of the Buffer

Run system commands over portions of the buffer using the ! operator. Use this operator to sort lines in the buffer or to reformat portions of the buffer. Try typing in a list of random words, one per line and ending them with a blank line. Back up to the beginning of the list and then give the **!}sort** command. This says to sort the next paragraph of material, and the blank line ends a paragraph.

Commands for Editing Lisp

If you are editing a LISP program, set the lisp option by typing

```
:se lisp
```

This changes the (and) commands to move backward and forward over s-expressions. The { and } commands are like (and) but do not stop at atoms. These can be used to skip to the next list or through a comment quickly.

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

There is another option that is useful for typing in LISP, the showmatch option. Try setting it with

```
:se sm
```

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

The editor also has an operator to realign existing lines as though they had been typed in with lisp and autoindent set. This is the = operator. Try the **=%** command at the beginning of a function. This will realign all the lines of the function declaration.

When you are editing LISP, the [[and]] advance and retreat to lines beginning with a (and are useful for dealing with entire function definitions.

Macros

The **vi** editor has a macro facility that lets you create a macro so that when you enter a single keystroke the editor will act as though you had entered a longer sequence of keystrokes. You can do this if you find yourself typing the same sequence of commands (keystrokes) repeatedly.

There are two types of macros:

- One type is where you put the macro body in a buffer register, such as *x*. You can then type @*x* to invoke the macro. The @ may be followed by another @ to repeat the last macro.
- You can use the **map** command from the **vi** editor (typically in your EXINIT) with a command of the form:

  ```
  :map lhs rhs
  ```

  mapping lhs into rhs. There are restrictions: lhs should be one keystroke (either one character or one function key). It must be entered within one second (unless notimeout is set, in which case you can type it as slowly as you wish, and **vi** will wait for you to finish it before it echoes anything). The lhs can be no longer than ten characters, the rhs no longer than 100. To get a space, tab, or newline into lhs or rhs you should escape them with a CODE-**v** (it may be necessary to double the CODE-**v** if the **map** command is given inside **vi** rather than in **ex**). Spaces and tabs inside the rhs need not be escaped.

  Thus to make the q key write and exit the editor, give the command and then press CODE-**v** twice and the RETURN key twice:

  ```
  :map q :wq
  ```

  This means that whenever you type q, it will be as though you had typed the four characters :wqRETURN. A CODE-**v** is needed because without it pressing the RETURN key would end the : command rather than becoming part of the map definition. There are two CODE-**v**'s because from within **vi**, two CODE-**v**'s must be typed to get on. The first RETURN is part of the rhs, the second terminates the : command.

Macros can be deleted with

```
unmap lhs
```

If the lhs of a macro is #**0** through #**9**, this maps the particular function key instead of the two-character # sequence. So that terminals without function keys can access such definitions, the form #*x* will mean function key *x* on all terminals (and need not be typed within one second). The character # can be changed by using a macro in the usual way:

```
:map (Press CODE-v twice) i #
```

to use tab, for example. This will not affect the **map** command, which still uses #, but just the invocation from visual mode.

The **undo** command reverses an entire macro call as a unit if it made any changes.

Placing a ! after the word map causes the mapping to apply to text input mode rather than command mode. Thus, to arrange for CODE-**t** to be the same as four spaces, type

```
:map (Press CODE-t and then CODE-v) ////
```

where / is a blank. The CODE-**v** is necessary to prevent the blanks from being taken as white space between the lhs and rhs.

Word Abbreviation

A feature similar to macros in text input mode is word abbreviation. This allows you to type a short word and have it expanded into a longer word or words. The commands are

:abbreviate

or

:unabbreviate

and

:ab

or

:una

These commands have the same syntax as **:map**. For example:

```
:ab eecs Electrical Engineering and Computer Sciences
```

causes the word "eecs" to always be changed into the phrase "Electrical Engineering and Computer Sciences." Word abbreviation is different from macros in that only whole words are affected. If "eecs" were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke as it should be with a macro.

Abbreviations

The editor has a number of short commands that abbreviate longer commands we have introduced here.

Additional Information

Line Representation in the Display

The editor folds long logical lines onto many physical lines in the display. Commands that advance lines advance logical lines and will skip over all the segments of a line in one motion. The | command moves the cursor to a specific column and may be useful for getting near the middle of a long line to split it in half. Try **80|** on a line that is more than 80 columns long.

Note: *You can make long lines very easily by using J to join together short lines.*

The editor puts only full lines on the display; if there is not enough room on the display to fit a logical line, the editor leaves the physical line empty placing only an @ on the line as a place holder. When you delete lines on a dumb terminal, the editor will often clear just the lines to @ to save time (rather than rewriting the rest of the screen). You can always maximize the information on the screen by giving the CODE-r command.

If you wish, have the editor place line numbers before each line on the display. Give the command

```
:se nu
```

to enable this, and the command

```
:se nonu
```

to turn it off. You can have tabs represented as CODE-i and the ends of lines indicated with **$** by giving the command

```
:se list
```

The following command removes the display of tabs and ends of lines:

```
:se nolist
```

Lines consisting of only the character are displayed when the last line in the file is in the middle of the screen. These represent physical lines that are past the logical end of file.

Counts

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

new window size	: / ? [[ ]] ` '
scroll amount	CODE-d CODE-u
line/column number	z G |
repeat effect	most of the rest

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

This size is the size used when the editor clears and refills the screen after a search or other motion moves far from the edge of the current window. Commands that take a new window size as count all too often cause the screen to be redrawn. If you anticipate this but do not need as large a window as you are currently using, change the screen size by specifying the new size before these commands.

In any case, the number of lines used on the screen will expand if you move off the top with a **-** or similar command or off the bottom with a command such as pressing the RETURN key or CODE-**d**. The window will revert to the last specified size the next time it is cleared and refilled (but not by a CODE-**l** which just redraws the screen as it is).

The scroll commands CODE-**d** and CODE-**u** likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus

```
        10a +----(Press the GO key)
```

will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands that ignore any counts (such as CODE-**r**), the rest of the editor commands use a count to indicate a simple repetition of their effect. Thus **5w** advances five words on the current line, while **5** advances five lines. A very useful instance of a count as a repetition is a count given to the . command, which repeats the last changing command. If you do **dw** and the **3.**, you will delete first one and then three words. You can then delete two more words with the **2.** command.

More File Manipulation Commands

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

:w	write back changes
:wq	write and quit
:x	write (if necessary) and quit (same as ZZ).
:e *filename*	edit *filename*
:e!	re-edit, discarding changes
:e + *filename*	edit, starting at end
:e + *n*	edit, starting at line n
:e #	edit alternate file
:w *filename*	write *filename*
:w! *filename*	overwrite *filename*
:*x,y*w *filename*	write lines x through y to *filename*
:r *filename*	read *filename* into buffer
:r!*command*	read output of *command* into buffer
:n	edit next file in argument list
:n!	edit next file, discarding changes to current
:n *args*	specify new argument list
:ta *tag*	edit file containing tag *tag*, at *tag*

All of these commands are followed by pressing either the RETURN or the GO key. The most basic commands are **:w** and **:e**. A normal editing session on a single file will end with a **ZZ** command. If you are editing for a long period of time, give **:w** commands occasionally after major amounts of editing, and then finish with a **ZZ**. When you edit more than one file, finish with a **:w** and start editing a new file by giving a **:e** command or set autowrite and use **:n**.

If you make changes to the editor copy of a file but do not wish to write them back, then you must give an ! after the command you would otherwise use; this forces the editor to discard any changes you have made. Use this carefully.

The :**e** command can be given a + argument to start at the end of the file or a +*n* argument to start at line *n*. In actuality, *n* may be any editor command not containing a space, usually a scan like +*/pat* or +*?pat*. In forming new names to the **e** command, use the character **%** which is replaced by the current file name or the character # which is replaced by the alternate file name. The alternate file name is generally the last name typed other than the current file. Thus, if you try to do a :**e** and get a diagnostic that you have not written into the file, give a :**w** command and then a :**e** # command to redo the previous :**e**.

Write part of the buffer to a file by finding the lines that bound the range to be written using CODE-**g** to obtain the line numbers. Enter these numbers after the : and before the **w**, separated by ,'s. You can also mark these lines with **m** and then use an address of the form *'x, 'y* on the **w** command.

Read another file into the buffer after the current line by using the :**r** command. You can similarly read in the output from a command, just use the :**r** *!cmd* instead of a file name.

If you wish to edit a set of files in succession, give all the names on the command line and then edit each one in turn using the command :**n**. It is also possible to respecify the list of files to be edited by giving the :**n** command a list of file names or a pattern to be expanded as you would have given it on the initial **vi** command.

If you are editing large programs, you will find the :**ta** command very useful. It utilizes a data base of function names and their locations that can be created by programs such as ctags, to quickly find a function whose name you give. If the :**ta** command will require the editor to switch files, then you must :**w** or abandon any changes before switching. Repeat the :**ta** command without any arguments to look for the same tag again.

More About Searching for Strings

When you are searching for strings in the file with / or **?**, the editor normally places you at the next or previous occurrence of the string. If you are using an operator such as **d**, **c**, or **y**, you may well wish to affect lines up to the line before the line containing the pattern. You can give a search of the form

```
/pat/ -n
```

to refer to the *n*'th line before the next line containing *pat*, or you can use + instead of - to refer to the lines after the one containing *pat*. If you do not give a line offset, the editor will affect characters up to the match place rather than whole lines; thus use +**0** to affect to the line that matches.

You can have the editor ignore the case of words in the searches it does by giving the command

```
:se ic
```

The command

```
:se noic
```

turns this off.

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

```
set nomagic
```

in your EXINIT. In this case, only the characters ˆ and **$** are special in patterns. The character \ is also then special (as it is most everywhere in the system) and may be used to get at the extended pattern-matching facility. It is also necessary to use a \ before a / in a forward scan or a **?** in a backward scan. The following gives the extended forms when magic is set:

^	At beginning of pattern, matches beginning of line.
$	At end of pattern, matches end of line.
.	Matches any character.
\<	Matches the beginning of a word.
\>	Matches the end of a word.
[*string*]	Matches any single character in *string.*
[^*string*]	Matches any single character not in *string.*
[*x-y*]	Matches any character between *x* and *y.*
*	Matches any number of the preceding pattern.

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

More About Input Mode

There are a number of characters you can use to make corrections during text input mode. These are summarized in the following table:

new window size	: / ? [[ ]] ` '
scroll amount	CODE-**d** CODE-**u**
line/column number	**z G |**
repeat effect	most of the rest
CODE-**h**	Deletes the last input character.
CODE-**w**	Deletes the last input word line.
GO	Ends an insertion.
DELETE	Interrupt an insertion, terminating it abnormally starts a new line.
CODE-**d**	Backtabs over autoindent.
0CODE-**d**	Kills all the autoindent.
^CODE-**d**	Same as 0CODE-d, but restores indent next line.
CODE-**v**	Quotes the next non-printing character into the file.

The most usual way of making corrections to input is by typing CODE-**h** to correct a single character or by typing one or more CODE-**w**'s to back over incorrect words. If you use # as the erase character in the normal system, it will work like CODE-**h**.

Your system kill character, normally @, CODE-**x** or CODE-**u**, will erase all the input given on the current line. In general, you can neither erase input back around a line boundary nor can you erase characters you did not insert with this insertion command. To make corrections on the previous line after a new line has been started, press the GO key to end the insertion, move over and make the correction, and then return to where you were to continue. The command **A** that appends at the end of the current line is often useful for continuing.

If you wish to type in the erase or kill character (such as # or @), then you must precede it with a \ just as you would do at the normal system command level. A more general way of typing nonprinting characters into the file is to precede them with a CODE-**v**. The CODE-**v** echoes as a ^ character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact, you may type any character; and it will be inserted into the file at that point.

Note: *This is not quite true. The implementation of the editor does not allow the NULL (CODE-@) character to appear in files. Also the LINEFEED (CODE-**j**) character is used by the editor to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ^ before you type the character. In fact, the editor will treat a following letter as a request for the corresponding control character. This is the only way to type CODE-**s** or CODE-**q** since the system normally uses them to suspend and resume output and never gives them to the editor to process.*

If you are using autoindent, backtab over the indent that it supplies by typing a CODE-**d**. This backs up to a shiftwidth boundary. This works only immediately after the supplied autoindent.

When using autoindent, you may wish to place a label at the left margin of a line. The way to do this easily is to type ^ and then CODE-**d**. The editor will move the cursor to the left margin for one line and restore the previous indent on the next. You can also type a **0** followed immediately by a CODE-**d** if you wish to kill all the indent and not have it come back on the next line.

Uppercase Only Terminate

If your terminal has only uppercase, you can still use **vi** by using the normal system convention for typing on such a terminal. Characters that you normally type are converted to lowercase, and you can type uppercase letters by preceding them with a \. The characters

```
{   }  |  `
```

are not available on such terminals, but you can escape them as

```
\(^\)\!\'.
```

These characters are represented on the display in the same way they are typed (the \ character you give will not echo until you type another key).

Relation Between vi and ex Editors

The **vi** editor is actually one mode of editing within the editor **ex**. When you are running **vi**, you can escape to the line-oriented editor of ex by giving the command **Q**. All of the : commands introduced above are available in **ex**. Likewise, most **ex** commands can be invoked from **vi** using :. Just give them without the : and then press the RETURN key.

In rare instances, an internal error may occur in **vi**. In this case, you will get a diagnostic and be left in the command mode of **ex**. You can then save your work and quit if you wish by giving a command **x** after the : that **ex** prompts you with, or you can reenter **vi** by giving **ex** a **vi** command.

There are a number of things that you can do more easily in **ex** than in **vi**. Systematic changes in line-oriented material are particularly easy. You can read the advanced editing documents for the editor **ed** to find out a lot more about this style of editing. Experienced users often mix their use of **ex** command mode and **vi** command mode to speed the work they are doing.

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

If you are on a hardcopy terminal or a terminal that does not have a cursor that can move off the bottom line, you can still use the command set of **vi**, but in a different mode. When you give a **vi** command, the editor will tell you that it is using open mode. This name comes from the open command in **ex**, which is used to get into the same mode.

The only difference between visual mode and open mode is the way the text is displayed.

In open mode, the editor uses a single line window into the file and moves backward and forward in the file causing new lines to be displayed always below the current line. Two commands of **vi** work differently in open mode: **z** and CODE-**r**. The **z** command does not take parameters, but rather draws a window of context around the current line and then returns to the current line.

If you are on a hardcopy terminal, the CODE-**r** command will retype the current line. On such terminals, the editor normally uses two lines to represent the current line. The first line is a copy of the line as you started to edit it, and you work on the line below this line. When you delete characters, the editor types a number of \'s to show the characters deleted. The editor also reprints the current line soon after such changes so that you can see what the line looks like again.

It is sometimes useful to use this mode on very slow terminals that can support **vi** in the full screen mode. You can do this by entering **ex** and using an open mode command.

Character Functions Summary

This summary shows the uses that the **vi** editor makes of each character. Characters are presented in their order in the ASCII character set: control characters first, most special characters, digits, uppercase characters, and then lowercase characters.

Each character is defined with a meaning it has as a command and any meaning it has during an insert. If it has meaning only as a command, then only this is discussed.

CODE-@	Not a command character. If typed as the first character of an insertion, it is replaced with the last text inserted; and the insert terminates. Only 128 characters are saved from the last insert; if more characters have been inserted, the mechanism is not available. A CODE-@ annot be part of the file due to the editor implementation.
CODE-a	Unused.
CODE-b	Backward window. A count specifies repetition. Two lines of continuity are kept if possible.
CODE-c	Unused.
CODE-d	As a command, it scrolls down a halfwindow of text. A count gives the number of (logical) lines to scroll and is remembered for future CODE-d and CODE-u commands. During an insert, it backtabs over autoindent white space at the beginning of a line. This white space cannot be backspaced over.
CODE-e	Exposes one more line below the current screen in the file, leaving the cursor where it is if possible.
CODE-f	Forward window. A count specifies repetition. Two lines of continuity are kept if possible
CODE-g	Equivalent to :f, printing the current file name, whether it has been modified, the current line number, the number of lines in the file, and the percent of the way through the file.
CODE-h **BACKSPACE**	Same as left arrow (see **h**). During an insert, it eliminates the last input character backing over it but not erasing it. The character remains so you can see what you typed if you wish to type something slightly different.

CODE-i **TAB**	Not a command character. When inserted, it prints as some number of spaces. When the cursor is at a tab character, it rests at the last of the spaces that represent the tab. The spacing of tab stops is controlled by the tabstop option.
CODE-j	Same as Down arrow. It moves the cursor one line down in the same column. If the position does not exist, **vi** comes as close as possible to the same column. Synonyms include **j** and CODE-**n**.
CODE-k	Unused.
CODE-l	The ASCII formfeed character that causes the screen to be cleared and redrawn. It is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt.
CODE-m ()	A carriage return advances to the next line, to the first nonwhite position in the line. Given a count, it advances that many lines. During an insert, a causes the insert to continue onto another line.
CODE-n	Same as Down Arrow. It moves the cursor one line down in the same column. If the position does not exist, **vi** comes as close as possible to the same column. Synonyms include **j** and CODE-**j**.
CODE-o	Unused.
CODE-p	Same as Up Arrow. It moves the cursor one line up. A synonym is **k**.
CODE-q	Not a command character. In text input mode, CODE-**q** quotes the next character, the same as CODE-**v**, except that some teletype drivers will eat the CODE-**q** so that the editor never sees it.
CODE-r	Redraws the current screen eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in open mode, retypes the current line.
CODE-s	Unused. Some teletype drivers use CODE-**s** to suspend output until CODE-**q** is invoked.
t	Not a command character. During an insert with autoindent set and at the beginning of the line, it inserts shiftwidth white space.

CODE-u	Scrolls the screen up inverting CODE-**d** which scrolls down. A count gives the number of (logical) lines to scroll and is remembered for future CODE-**d** and CODE-**u** commands. The previous scroll amount is common to both. On a dumb terminal, CODE-**u** will often necessitate clearing and redrawing the screen further back in the file.
CODE-v	Not a command character. In text input mode, it quotes the next character so that it is possible to insert nonprinting and special characters into the file.
CODE-w	Not a command character. During an insert, it backs up as b would in command mode; the deleted characters remain on the display (see CODE-**h**).
CODE-x	Unused.
CODE-y	Exposes one more line above the current screen leaving the cursor where it is if possible. There is no mnemonic value for this key; however, it is next to CODE-**u**.
CODE-z	Unused.
CODE-[**GO**	Cancels a partially formed command (such as a **z** when no following character has yet been given), terminates inputs on the last line (read by commands such as :, /, and ?), and ends insertions of new text into the buffer. If the GO key is pressed when quiescent in command state, the editor rings the bell or flashes the screen. You can thus press the GO key if you do not know what is happening until the editor rings the bell. If you do not know if you are in insert mode, press the GO key and then **a** and then type the material to be input, the material will be inserted correctly whether or not you were in insert mode when you started.
CODE-\	Unused.
CODE-]	Searches for the word that is after the cursor as a tag. It is equivalent to typing **:ta**, this word, and then pressing the RETURN key.
CODE-CODE	Equivalent to **:e** #, returning to the previous position in the last edited file, or editing a file that you specified if you got a "No write since last change" diagnostic and do not want to have to type the file name again. You have to do a **:w** before CODE-CODE will work in this case. If you do not wish to write the file, enter **:e!** # instead.
SPACEBAR	Same as Right Arrow (see **l**).

!	An operator that processes lines from the buffer with reformatting commands. Follow ! with the object to be processed, and then the command name terminated by pressing the RETURN key. Doubling ! and preceding it by a count causes count lines to be filtered; otherwise, the count is passed on to the object after the !. Thus 2!{*fmt* reformats the next two paragraphs by running them through the program *fmt*. To read a file or the output of a command into the buffer use :r. To simply execute a command use :!.
"	Precedes a named buffer specification. There are named buffers 1-9 used for saving deleted text and named buffers a-z into which you can place text.
#	The macro character, when followed by a number, will substitute for a function key on terminals without function keys. In text input mode if this is your erase character, it will delete the last character you typed and must be preceded with a \ to insert it since it normally backs over the last input character you gave.
$	Moves to the end of the current line. If the :**se** list command is used, then the end of each line will be shown by printing a **$** after the end of the displayed text in the line. When a count is used, the cursor advances to the end of the line following the count. For example, **2$** advances the cursor to the end of the following line.
%	Moves to the parenthesis or brace that precedes or follows the parenthesis or brace at the current cursor position.
&	A synonym for :&, analogous to the **ex** & command.
'	When followed by a ', the cursor returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a nonrelative way. When followed by a letter (a-z), it returns to the line that was marked with this letter with an m command at the first nonwhite character in the line. When used with an operator such as **d**, the operation takes place over complete lines; if you use `, the operation takes place from the exact marked place to the current cursor position within the line.
(	Retreats to the beginning of a sentence or to the beginning of a LISP **s**-expression if the lisp option is set. A sentence ends at a ., !, or ? followed by either the end of a line or by two spaces. Any number of closing characters ( ),], ", and ') may appear after the ., !, or ?, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see { and [[). A count advances that many sentences.

)	Advances to the beginning of a sentence. A count repeats the effect. See (for the definition of a sentence.
*	Unused.
+	Same as when used as a command.
,	Reverse of the last **f**, **F**, **t**, or **T** command, looking the other way in the current line. Especially useful after typing too many ; characters. A count repeats the search.
-	Retreats to the previous line at the first nonwhite character. This is the inverse of + and pressing the RETURN key. If the line moved to is not on the screen, the screen is scrolled or cleared and redrawn. If a large amount of scrolling would be required, the screen is also cleared and redrawn with the current line at the center.
.	Repeats the last command that changed the buffer. Especially useful when deleting words or lines; you can delete some words/lines and then type . to delete more and more words/lines. Given a count, it passes it on to the command being repeated. Thus after a **2dw**, a **3**. deletes three words.
/	Reads a string from the last line on the screen and scans forward for the next occurrence of this string. The search begins when you press the RETURN key to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress; the search may be terminated by pressing the DELETE key or by backspacing when at the beginning of the bottom line and returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer. When used with an operator, the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern, you can force whole lines to be affected. To do this, give a pattern with a closing / and then an offset +*n* or -*n*. To include the / character in the search string, you must escape it with a preceding \. A ^ at the beginning of the pattern forces the match to occur at the beginning of a line only; this speeds the search. A **$** at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available. Unless you set nomagic in your .exrc file, you will have to precede the characters ., [, *, and ~ in the search pattern with a **0** to get them to work as you would expect.
0	Moves to the first character on the current line. Also used, when forming numbers, after an initial 1-9.

1-9	Used to form numeric arguments to commands.
:	A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with a , and the command is then executed. Return to where you were by pressing the DELETE key if you type : accidentally.
;	Repeats the last single character find that used **f**, **F**, **t**, or **T**. A count iterates the basic scan.
<	An operator that shifts lines left one shiftwidth, normally eight spaces. Like all operators, it affects lines when repeated, as in < <. Counts are passed through to the basic object, thus **3**<< shifts three lines.
=	Reindents lines for LISP, as though they were typed in with lisp and autoindent set.
>	An operator that shifts lines right one shiftwidth, normally eight spaces. Affects lines when repeated as in >>. Counts repeat the basic object.
?	Scans backwards, the opposite of /. See the / description for details on scanning.
@	A macro character. If this is the kill character, you must escape it with a \ to type it in during text input mode since it normally backs over the input given on the current line.
A	Appends at the end of line, a synonym for **$a**.
B	Backs up a word, where words are composed of nonblank sequences, placing the cursor at the beginning of the word. A count repeats the effect.
C	Changes the rest of the text on the current line; a synonym for **c$**.
D	Deletes the rest of the text on the current line; a synonym for **d$**.
E	Moves forward to the end of a word, defined as blanks and nonblanks, like **B** and **W**. A count repeats the effect.
F	Finds a single following character, backward in the current line. A count repeats this search that many times.

G	Goes to the line number given as preceding argument or the end of the file if no preceding count is given. The screen is redrawn with the new current line in the center if necessary.
H	Same as Home arrow. Homes the cursor to the top line on the screen. If a count is given, then the cursor is moved to the count's line on the screen. In any case the cursor is moved to the first nonwhite character on the line. If used as the target of an operator, full lines are affected.
I	Inserts at the beginning of a line; a synonym for CODE-i.
J	Joins lines together, supplying appropriate white space: one space between words, two spaces after a ., and no spaces at all if the first character of the joined on line is **)**. A count causes that many lines to be joined rather than the default two.
K	Unused.
L	Moves the cursor to the first nonwhite character of the last line on the screen. With a line count number, moves the cursor to the first nonwhite character of the indicated line from the bottom. Operators affect whole lines when used with L.
M	Moves the cursor to the middle line on the screen at the first nonwhite position on the line.
N	Scans for the next match of the last pattern given to / or **?** but in the reverse direction; this is the reverse of **n**.
O	Opens a new line above the current line and inputs text until the GO key is pressed. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete as the slowopen option works better.
P	Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines; otherwise, the text is inserted between the characters before and at the cursor. The **P** character may be preceded by a named buffer specification **"x** to retrieve the contents of the buffer; buffers 1-9 contain deleted material, buffers a-z are available for general use.

Q	Quits from **vi** to **ex** command mode. In this mode, whole lines form commands and end when the RETURN key is pressed. You can give all the : commands; the editor supplies the : as a prompt.
R	Replaces characters on the screen with characters you type (overlay fashion). Terminates when the GO key is pressed.
S	Changes whole lines; a synonym for **cc**. A count substitutes for that many lines. The lines are saved in the numeric buffers and erased on the screen before the substitution begins.
T	Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as **d**.
U	Restores the current line to its state before you started changing it.
V	Unused.
W	Moves forward to the beginning of a word in the current line where words are defined as sequences of blank/nonblank characters. A count repeats the effect.
X	Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.
Y	Yanks a copy of the current line into the unnamed buffer to be put back by a later **p** or **P**; a synonym for **yy**. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer.
ZZ	Exits the editor (same as :x). If any changes have been made, the buffer is written out to the current file. Then the editor quits.
[[	Backs up the previous section boundary. A section begins at each macro in the sections options, normally a .NH or .SH, and also at lines that start with a formfeed CODE-l. Lines beginning with { also stop [[; this makes it useful for looking backwards, a function at a time, in C programs. If the option lisp is set, stops at each (at the beginning of a line and is thus useful for moving backwards at the top level LISP objects.

\	Unused.
]]	Forward to a section boundary. See [[for a definition.
^	Moves to the first nonwhite position on the current line.
_	Unused.
`	When followed by a `, returns to the previous context. The previous context is set whenever the current line is moved in a nonrelative way. When followed by a letter a-z, returns to the position that was marked with this letter with an **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; if you use ', the operation takes place over complete lines.
a	Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates when the GO key is pressed.
b	Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics or a sequence of special characters. A count repeats the effect.
c	An operator that changes the following object replacing it with the following input text until the GO key is pressed. If more than part of a single line is affected, the text that is changed away is saved in the numeric named buffers. If only part of the current line is affected, the last character to be changed away is marked with a **$**. A count causes that many objects to be affected, thus both **3c)** and **c3)** change the following three sentences.
d	An operator that deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w**.
e	Advances to the end of the next word, defined as for **b** and **w**. A count repeats the effect.
f	Finds the first instance of the next character following the cursor on the current line. A count repeats the find.
g	Unused.

h	Same as Left Arrow. Moves the cursor one character to the left. Like the other arrow keys, either h, the Left Arrow key, or one of the synonyms (CODE-**h**) has the same effect. A count repeats the effect.
i	Inserts text before the cursor; otherwise, like **a**.
j	Same as Down Arrow. Moves the cursor one line down in the same column. If the position does not exist, **vi** comes as close as possible to the same column. Synonyms include CODE-**j** and CODE-**n**.
k	Same as Up Arrow. Moves the cursor one line up. CODE-**p** is a synonym.
l	Same as Right Arrow. Moves the cursor one character to the right. SPACEBAR is a synonym.
m	Marks the current position of the cursor in the mark register that is specified by the next character a-z. Return to this position or use with an operator using ` or '.
n	Repeats the last / or **?** scanning commands.
o	Opens new lines below the current line; otherwise, like **O**.
p	Puts text after/below the cursor; otherwise, like **P**.
q	Unused.
r	Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R** above, which is usually the more useful iteration of **r**.
s	Changes the single character under the cursor to the text that follows until the GO key is pressed; given a count, that many characters from the current line are changed. The last character to be changed is marked with **$** as in **c**.
t	Advances the cursor up to the character before the next character typed. Most useful with operator such as **d** and **c** to delete the characters up to a following character. You can use . to delete more if this does not delete enough the first time.
u	Undoes the last change made to the current buffer. If repeated, will alternate between these two states, thus is its own inverse. When used after an insert that inserted text on more than one line, the lines are saved in the numeric named buffers.

v	Unused.
w	Advances to the beginning of the next word, as defined by **b**.
x	Deletes the single character under the cursor. With a count, deletes that many characters forward from the cursor position, but only on the current line.
y	An operator that yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, **"x**, the text is placed in that buffer also. Text can be recovered by a later **p** or **P**.
z	Redraws the screen with the current line placed as specified by the following character:

RETURN	Specifies the top of the screen.
.	Specifies the center of the screen.
-	Specifies the bottom of the screen.

A count may be given after the **z** and before the following character to specify the new screen size for the redraw. A count before the **z** gives the number of the line to place in the center of the screen instead of the default current line.

{	Retreats to the beginning of the preceding paragraph. A paragraph begins at each macro in the paragraphs option, normally .IP, .LP, .PP, .QP and **.bp**. A paragraph also begins after a completely empty line and at each section boundary (see **[[**).
|	Places the cursor on the character in the column specified by the count.
}	Advances to the beginning of the next paragraph. See **{** for the definition of paragraph.
~	Switches character from lowercase to uppercase and vise versa.
CODE-? DELETE	Interrupts the editor, returning it to command-accepting state.

Text Editor (ex)

Starting the Text Editor

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

A command to enter the **ex** editor has the following format (brackets, [ ], indicate optional parameters).

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

- The most common case edits a single file with no options, for example:

  ex *filename*

- The **-** command line option suppresses all interactive-user feedback and is useful in processing editor scripts in command files.
- The **-v** option is equivalent to using **vi** rather than **ex**.
- The **-t** option is equivalent to an initial *tag* command, editing the file containing the *tag*, and positioning the editor at its definition.
- The -r option is used in recovering after an editor or system crash, retrieving the last saved version of the named file, or if no file is specified, typing a list of saved files.
- The **-l** option sets up for editing LISP setting the showmatch and lisp options.
- The **-w** option sets the default window size to *n* and is useful on dial-ups to start in small windows.

- The **-x** option causes **ex** to prompt for a *key*. The *key* is used to encrypt and decrypt the contents of the file. The file should already be encrypted using this same key.
- The **-R** option sets the readonly option at the start.
- The +*command* argument indicates that the editor should begin by executing the specified command. If *command* is omitted, then command defaults to $, positioning the editor at the last line of the first file. Other useful commands here are scanning patterns of the form */pat* or line numbers, for example, +**100** starting at line 100.
- The *filename* arguments indicate files to be edited.

Manipulating Files

Current File

The **ex** editor is normally editing the contents of a single file, whose name is recorded in the current file name. The **ex** editor performs all editing actions in a buffer (a temporary file) into which the text of the file is initially read. Changes made to the buffer have no effect on the file being edited until the buffer contents are written out to the file with a **write** command. After the buffer contents are written, the previous contents of the written file are no longer accessible. When a file is edited, its name becomes the current file name and its contents are read into the buffer.

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

Alternate File

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

File Name Expansion

File names within the editor may be specified using the normal shell expansion conventions. In addition, the character % in file names is replaced by the current file name and the character # by the alternate file name. This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an **edit** command after a "No write since last change" diagnostic message is received.

Multiple Files and Named Buffers

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

For saving blocks of text while editing and especially when editing more than one file, **ex** has a group of named buffers. These are similar to the normal buffer, except that only a limited number of operations are available on them. The buffers have names **a** through **z**. It is also possible to refer to **A** through **Z**; the uppercase buffers are the same as the lowercase but commands append to named buffers rather than replacing if uppercase names are used.

Read-Only Mode

It is possible to use **ex** in the read-only mode to look at files that you have no intention of modifying. This mode protects you from accidently overwriting the file. Read-only mode is on when the readonly option is set. It can be turned on with the **-R** command line option, by the **view** command line invocation, or by setting the readonly option. It can be cleared by setting the noreadonly mode.

It is possible to write, even while in read-only mode, by indicating that you really know what you are doing. You can write to a different file or use the ! form of write.

Exceptional Conditions

Errors and Interrupts

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

Recovering from Hang-ups and Crashes

If a hang-up signal is received and the buffer has been modified since it was last written or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second case) will attempt to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing at most a few lines of changes from the last point before the hang-up or editor crash.

To recover a file, use the **-r** option. For example, if you were editing the file resume, then you should change to the directory where you were when the crash occurred, giving the command

```
    ex -r resume
```

After checking that the retrieved file is good, write it over the previous contents of that file.

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

```
    ex -r
```

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

Editing Modes

The **ex** editor has five distinct modes.

- The primary mode is the command mode. Commands are entered in command mode when a : prompt is present and are executed each time a complete line is sent.
- In text input mode, the **ex** editor gathers input lines and places them in the file. The **append**, **insert**, and **change** commands use text input mode. No prompt is printed. This mode is left by typing a single period (.) alone at the beginning of a line and command mode resumes.

The last three modes are open mode, visual mode (entered by the commands of the same name), and text insertion mode (within open and visual modes).

- The open mode allows local editing operations to be performed on the text in the file. The **open** command displays one line at a time and can be used on any terminal.
- The visual mode allows local editing operations to be performed on the text in the file. The **visual** command works on cathode ray tube (CRT) terminals with random positioning cursors using the screen as a single window for file editing changes.

Command Structure

Most command names are English words (initial prefixes of the words are acceptable abbreviations). The ambiguity of abbreviations is resolved in favor of the more commonly used commands. As an example, the **substitute** command can be abbreviated **s**. The shortest available abbreviation for the **set** command is **se**.

Command Parameters

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

Some commands take other information or parameters, the information always being given after the command name. Examples would be option names in a **set** command (**set** *number*), a file name in an **edit** command, a regular expression in a substitute command, or a target address for a **copy** command (1,5 copy 25).

Command Variants

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

Flags after Commands

The characters #, **p**, and **l** may be placed after many commands (a **p** or **l** must be preceded by a blank or tab except in the single special case **dp**). In this case, the command abbreviated by these characters is executed after the command completes. Since **ex** normally prints the new current line after each change, **p** is rarely necessary. Any number of + or - characters may also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

Comments

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

Multiple Commands Per Line

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

Reporting Large Changes

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

Command Addressing

Addressing Primitives

.	The . represents the current line. Most commands leave the current line as the last line they affect. The default address for most commands is the current line, thus . is rarely used alone as an address.
n	The *n*th line in the editor buffer, lines being numbered sequentially from 1.
$	The last line in the buffer.
%	An abbreviation for **1,$**, the entire buffer.
+**n** -**n**	An offset relative to the current buffer line. The forms .+3, +3, and +++ are all equivalent; if the current line is line 100, they all address line 103.
/**pat**/	Scans forward for a line containing pat (a regular expression). Scans normally wraparound the end of the buffer. If all that is desired is to print the next line containing pat, then the trailing / may be omitted. If pat is omitted or explicitly empty, then the last regular expression specified is located. The form \/ scans using the last regular expression used in a scan. After a substitute, // scans using the substitute's regular expression.

?pat?	Scans backward for a line containing pat (a regular expression). Scans normally wraparound the end of the buffer. If all that is desired is to print the preceding line containing pat, then the trailing ? may be omitted. If pat is omitted or explicitly empty, then the last regular expression specified is located. The form \? scans using the last regular expression used in a scan. After a substitute, ?? scans using the substitute's regular expression.
'' 'x	Before each nonrelative motion of the current line ., the previous current line is marked with a tag, subsequently referred to as ''. This makes it easy to refer or return to this previous context. Marks may also be established by the **mark** command using single lowercase letters **x** and the marked lines referred to as **'x**.

Combining Addressing Primitives

Addresses to commands consist of a series of addressing primitives separated by , or ;. Such address lists are evaluated left-to-right. When , seperates addresses, the current line . is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer. Null address specifications are permitted in a list of addresses, the default in this case is the current line (.); thus, **,100** is equivalent to **.,100**. It is an error to give a prefix address to a command which expects none.

Command Descriptions

The following form is the format for all **ex** commands:

```
address command ! parameters count flags
```

All parts are optional; the degenerate case is the empty command that prints the next line in the file. For sanity with use from within visual mode, the **ex** editor ignores a . preceding any command. In the following command descriptions, the default addresses are shown in parentheses. Note that parentheses are not part of the command syntax.

abbreviate *word rhs* Abbreviation: **ab**

Add the named abbreviation to the current list. When in the input mode in **vi**, if *word* is typed as a complete word, it will be changed to *rhs*.

(.) **append** Abbreviation: **a**
text
.

Reads the input text and places it after the specified line. After the command, . addresses the last line input or the specified line if no lines were input. If address 0 (zero) is given, text is placed at the beginning of the buffer.

a!
text
.

The variant flag to append toggles the setting for the autoindent option during the input of *text*.

args

The members of the argument list are printed with the current argument delimited by [and].

(.,.) **change** *count* Abbreviation: **c**
text
.

Replaces the specified lines with the input *text*. The current line becomes the last line input; if no lines were input, it is left as for a **delete**.

c!
text
.

The variant toggles autoindent during the change.

(.,.) **copy** *addr flags* Abbreviation: **co**

A copy of the specified lines is placed after *addr*, which may be 0. The current line . addresses the last line of the copy. The command **t** is a synonym for **copy**.

(.,.) **delete** *buffer count flags* Abbreviation: **d**

Removes the specified lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line. If a named *buffer* is specified by giving a letter, the specified lines are saved in that buffer or appended to it if an uppercase letter is used.

ex *file* Abbreviation: **e**

Used to begin an editing session on a new file. The editor first checks to see if the current buffer has been modified since the last **write** command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file, and prints the new file name.

After insuring that this file is not a binary file (such as a directory), a block or character special file (other than /dev/tty), a terminal, or a binary or executable file (as indicated by the first word), the editor reads the file into its buffer. If the read of the file completes without error, the number of lines and characters read is typed. If there were any non-ASCII characters in the file they are stripped of their non- ASCII high bits, and any null characters in the file are discarded.

If none of these errors occurred, the file is considered edited. If the trailing newline character is missing from the last line of the input file, it will be supplied and a complaint will be issued. This command leaves the current line (.) at the last line read. If executed from within open or visual mode, the current line is initially the first line of the file.

e! *file*

The variant form suppresses the complaint about modifications having been made and not written from the editor buffer, thus discarding all changes which have been made before editing the new file.

e +*n file*

Causes the editor to begin at line *n* rather than at the last line; *n* may also be an editor command containing no spaces, for example, **+/pat**.

file Abbreviation: **f**

The **file** command displays:

the current file name.

whether it has been modified since the last **write** command.

whether it is 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.

In the rare case that the current file is not edited, this is noted also. In this case, you have to use the form **w!** to write to the file since the editor is not sure that a **write** command will not destroy a file unrelated to the current contents of the buffer.

file file

The current file name is changed to the file which is considered not edited.

(1,$) **global** */pat/cmds* Abbreviation: **g**

First marks each line among those specified which matches the given regular expression. Then the given command list is executed with . initially set to each marked line.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a \. If *cmds* (and possibly the trailing / delimiter) is omitted, each line matching *pat* is printed. The **append, insert,** and **change** commands and associated input are permitted; the . terminating input may be omitted if it would be on the last line of the command list. The **open** and **visual** commands are permitted in the command list and take input from the terminal.

The **global** command itself may not appear in *cmds*. The **undo** command is also not permitted there since **undo** instead can be used to reverse the entire **global** command. The options autoprint and autoindent are inhibited during a **global** command (and possibly the trailing / delimiter), and the value of the report option is temporarily infinite in deference to a report for the entire **global** command. Finally, the context mark ('') is set to the value of . before the **global** commands begin and is not changed during a **global** command except perhaps by an open or visual mode within the **global** command.

g! */pat/cmds* Abbreviation: **v**

The variant form of a **global** command runs cmds at each line not matching pat.

(.) **insert** Abbreviation: **i**
text
.

Places the given text before the specified line. The current line is left at the last line input. If there were no lines input it is left at the line before the addressed line. This command differs from **append** only in the placement of text.

i!
text
.

The variant toggles autoindent during the insert.

(.,.+1) **join** *count flags* Abbreviation: **j**

Places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character, two if there was **a** . at the end of the line, or none if the first following character is **a**). If there is already white space at the end of the line, the white space at the start of the next line will be discarded.

j!

The variant causes a simpler join with no white space processing. Characters in the lines are simply concatenated.

(.) **k** *x*

The **k** command is a synonym for **mark.** It does not require a blank or tab before the following letter.

(.,.) **list** *count flags*

Prints the specified lines in a more unambiguous way. Tabs are printed as CODE-**I** and the end of each line is marked with a trailing **$**. The current line is left at the last line printed.

map *lhs rhs*

The **map** command is used to define macros for use in visual mode. The lhs should be a single character or the #*n* sequence (for a digit) referring to function key *n.* When this character or function key is typed in visual mode, it will be as though the corresponding rhs has been typed. On terminals without function keys, type #*n.*

(.) **mark** *x*

Gives the specified line mark *x,* a single lowercase letter. The *x* must be preceded by a blank or a tab. The addressing form **'x** then addresses this line. The current line is not affected by this command.

(.,.) **move** *addr* Abbreviation: **m**

The **move** command repositions the specified lines to be after *addr.* The first of the moved lines becomes the current line.

next Abbreviation: **n**

The next file from the command line argument list is edited.

n!

The variant suppresses warnings about the modifications to the buffer not having been written out discarding (irretrievably) any changes made.

n *filelist*
n +*command filelist*

The specified *filelist* is expanded and the resulting list replaces the current argument list. The first file in the new list is then edited. If *command* is given (it must contain no spaces), then it is executed after editing the first such file.

(.,.) *number count flags* Abbreviation: # or **nu**

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

(.) **open** *flags* Abbreviation: **o**
(.) **open** */pat/flags*

Enters intraline editing open mode at each addressed line. If pat is given, then the cursor will be placed initially at the beginning of the string matched by the pattern. To exit this mode, use **Q**.

preserve

The current editor buffer is saved as though the system has just crashed. This command is for use only in emergencies when a **write** command has resulted in an error and you do not know how to save your work. After a **preserve**, seek help.

(.,.)**print** *count* Abbreviation: **p** or **P**

Prints the specified lines with nonprinting characters printed as control characters CODE-**x**; delete (octal 177) is represented as CODE-?. The current line is left at the last line printed.

(.)**put** *buffer* Abbreviation: **pu**

Puts back previously deleted or yanked lines. Normally used with **delete** to effect movement of lines or with **yank** to effect duplication of lines. If no buffer is specified, then the last deleted or yanked text is restored. No modifying commands may intervene between the **delete** or **yank** and the **put**, nor may lines be moved between files without using a named buffer. By using a named buffer, text may be restored that was saved at any previous time.

quit Abbreviation: **q**

Causes the **ex** editor to terminate. No automatic write of the editor buffer to a file is performed. However, **ex** issues a warning message if the file has changed since the last **write** command was issued and does not **quit** (the **ex** editor will also issue a diagnostic if there are more files in the argument list). Normally, you will wish to save your changes and you should give a **write** command. If you wish to discard them, use the **q!** command variant.

q!

Quits from the editor discarding changes to the buffer without complaint.

(.)**read** *filename* Abbreviation: **r**

Places a copy of the text of the given file in the editing buffer after the specified line. If no *filename* is given, the current file name is used. The current file name is not changed unless there is none, in which case *filename* becomes the current name. The sensibility restrictions of the **edit** command apply here also. If the file buffer is empty and if there is no current name, then the **ex** editor treats this as an **edit** command.

Address 0 (zero) is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the **edit** command when the **read** successfully terminates. After a **read** command, the current line is the last line read. However, within open and visual mode, the current line is set to the first line read rather than the last.

(.)**read** !*command*

Reads the output of the command **command** into the buffer after the specified line. This is not a variant form of the command. It is a read specifying a command rather than a filename. A blank or tab before the ! is mandatory.

recover *filename*

Recovers *filename* from the system save area. It is used after an accidental hang-up of the phone, a system crash (the system saves a copy of the file you are editing only if you have made changes to the file), or a **preserve** command. When you use the **preserve** command, you will be notified by mail when a file is saved.

rewind Abbreviation: **rew**

The argument list is rewound, and the first file in the list is edited.

rew!

Rewinds the argument list discarding any changes made to the current buffer.

set *parameter*

With no arguments, prints those options whose values have been changed from their defaults; with parameter **all**, it prints all of the option values.

Giving an option name followed by a **?** causes the current value of that option to be printed. The **?** is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form "set option" to turn them on or "set nooption" to turn them off. String and numeric options are assigned via the form "set option=value."

More than one parameter may be given to set; they are interpreted left-to-right.

shell Abbreviation: **sh**

A new shell is created. When it terminates, editing resumes.

source *filename* Abbreviation: **so**

Reads and executes commands from the specified file. The **source** commands may be nested.

(.,.)**substitute** */pat/repl/options count flags* Abbreviation: **s**

On each specified line, the first instance of pattern pat is replaced by replacement pattern repl. If the global indicator option character **g** appears, then all instances are substituted. If the confirm indication character **c** appears, then before each substitution, the line to be substituted is typed with the string to be substituted marked with ^ characters. By typing a **y**, one can cause the substitution to be performed; any other input causes no change to take place. After a **substitute** command, the current line is the last line substituted.

Lines may be split by substituting newline characters into them. The newline in *repl* must be escaped by preceding it with a \. Other metacharacters available in *pat* and *repl* are described below.

(.,.) **substitute** *options count flags* Abbreviation: ***s***

If *pat* and *repl* are omitted, then the last substitution is repeated. This is a synonym for the **&** command.

(.,.) **t** *addr flags*

The **t** command is a synonym for **copy**.

ta *tag*

The focus of editing switches to the location of *tag*, to a different line in the current file where it is defined, or if necessary to another file. If you have modified the current file before giving a **tag** command, you must write it out; giving another **tag** command, specifying no *tag* will reuse the previous *tag*.

The *tag* file is normally created by a program such as ctags and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form which can be used by the editor to find the tab. This field is usually a contextual scan using */pat/* to be immune to minor changes in the file. Such scans are always performed as if nomagic was set.

Names in the *tag* file must be sorted alphabetically.

unabbreviate *word* Abbreviation: **una**

Delete *word* from the list of abbreviations.

undo Abbreviation: **u**

Reverses the changes made in the buffer by the last buffer editing command.

Note that **global** commands are considered a single command for the purpose of **undo** (as are **open** and **visual**). Also, the commands **write** and **edit** which interact with the file system cannot be undone; **undo** is its own inverse.

The **undo** command always marks the previous value of the current line (.) as ''. After an **undo** command, the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect such as **global** and **visual**, the current line regains its precommand value after an **undo**.

unmap *lhs*

The macro expansion associated by **map** for *lhs* is removed.

(1,$) *v/pat/cmds*

A synonym for the **global** command variant **g!**, running the specified cmds on each line which does not match pat.

version Abbreviation: **ve**

Prints the current version number of the editor as well as the date the editor was last changed.

(.) **visual** *type count flags* Abbreviation: **vi**

Enters visual mode at the specified line. The *type* argument is optional and may be –, ^, or . as in the **z** command to specify the placement of the specified line on the screen. By default, if *type* is omitted, the specified line is placed as the first on the screen. A *count* specifies an initial window size; the default is the value of the option window. To exit this mode, type **Q**.

visual *filename*
visual *+n filename*

From visual mode, this command is the same as edit.

(1,$) write *filename* Abbreviation: **w**

Write changes made back to *filename*, printing the number of lines and characters written. Normally, *filename* is omitted and the text goes back where it came from. If *filename* is specified, then text will be written to that file.

The editor writes to a file only if it is the current file and is edited, if the file does not exist, or if the file is actually a teletype (/dev/tty, /dev/null). Otherwise, you must give the variant form **w!** to force the write.

If the file does not exist, it is created. The current file name is changed only if there is no current file name; the current line is never changed.

If an error occurs while writing the current and edited file, the editor considers that there has been no write since last change even if the buffer had not previously been modified.

(1,$) write>>*filename* Abbreviation: **w>>**

Writes the buffer contents at the end of an existing file.

w! *filename*

Overrides the checking of the normal **write** command and will write to any file which the system permits.

(1,$) w !*command*

Writes the specified lines into *command.*

Note that there is a difference between **w!** which overrides checks and **w !** which writes to a command.

wq *filename*

Like a **write** and then a **quit** command.

wq! *filename*

The variant overrides checking on the sensibility of the **write** command, as **w!** does.

xit *filename*

If any changes have been made and not written, writes the buffer out. Then, in any case, quits.

(.,.)yank *buffer count* Abbreviation: **ya**

Places the specified lines in the named buffer for later retrieval via **put**. If no buffer name is specified, the lines go to a more volatile place (see the **put** command description).

(.+1) z *count*

Prints the next *count* lines (default window).

(.) **z** *type count*

Prints a window of text with the specified line at the top. If *type* is **-**, the line is placed at the bottom, **a .** causes the line to be placed in the center.

Note that forms **z=** and **z^** also exist; **z=** places the current line in the center, surrounds it with lines of - characters, and leaves the current line at this line. The form **z^**prints the window before **z-** would. The characters **+**, ^ and **-** may be repeated for cumulative effect.

A count gives the number of lines to be displayed rather than double the number specified by the scroll option. On a CRT, the screen is cleared before display begins unless a count which is less than the screen size is given. The current line is left at the last line printed.

! *command*

The remainder of the line after the ! character is sent to a shell to be executed. Within the text of *command*, the characters % and # are expanded as in file names; and the ! character is replaced with the text of the previous command. Thus, in particular, !! repeats the last such shell escape. If any such expansion is performed, the expanded line will be echoed. The current line is unchanged by this command.

If there has been "no write" of the buffer contents since the last change to the editing buffer, then as a warning, a diagnostic message will be printed before the command is executed. A single ! is printed when the command completes.

(*addr, addr*) ! *command*

Takes the specified address range and supplies it as standard input to *command*. The resulting output then replaces the input lines.

($) =

Prints the line number of the addressed line. The current line is unchanged.

(.,.) > *count flags*
(.,.) < *count flags*

Perform intelligent shifting on the specified lines: < shifts left and > shifts right. The quantity of shift is determined by the shiftwidth option and the repetition of the specification character. Only white space (blanks and tabs) is shifted. No nonwhite characters are discarded in a left shift. The current line becomes the last line that changed due to the shifting.

d

An end of file from a terminal input scrolls through the file. The scroll option specifies the size of the scroll, normally a half screen of text.

(.+1,.+1)
(.+1,.+1)|

An address alone causes the addressed lines to be printed. A blank line prints the next line in the file.

(.,.) **&** *options count flags*

Repeats the previous substitute command.

(.,.) ~ *options count flags*

Replaces the previous regular expression with the previous replacement pattern from a substitution.

Regular Expressions and Substitute Replacement Patterns

Regular Expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be matched by the regular expression. The **ex** editor remembers two previous regular expressions: the previous regular expression used in a substitute command and the previous regular expression used elsewhere (referred to as the previous scanning regular expression). The previous regular expression can always be referred to by a null **re** (// or ??).

Magic and Nomagic

Regular expressions allowed by the **ex** editor are constructed in one of two ways depending on the setting of the magic option. The **ex** and **vi** editors default setting of magic gives quick access to a powerful set of regular expression metacharacters. The disadvantage of magic is that the user must remember that these metacharacters are magic and precede them with the \ character to use them as ordinary characters. With the nomagic option, regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the (now) ordinary character with a \.

Note: *The \ is always a metacharacter.*

The remainder of the discussion of regular expressions assumes that the setting of this option is magic. To discern what is true with nomagic, it suffices to remember that the only special characters in this case will be ^ at the beginning of a regular expression, **$** at the end of a regular expression, and \. With nomagic the characters ~ and **&** also lose their special meanings to the replacement pattern of a substitute.

Basic Regular Expression Summary

The following basic constructs are used to construct magic mode regular expressions.

char	An ordinary character matches itself. The characters ^ at the beginning of a line, **$** at the end of line, * as any character other than the first, ., \, [, and ~ are not ordinary characters and must be escaped (preceded) by \ to be treated as such.
^	At the beginning of a pattern forces the match to succeed only at the beginning of a line.
$	At the end of a regular expression forces the match to succeed only at the end of the line.
.	Matches any single character except the newline character.
\<	Forces a match to occur only at the beginning of a variable or word; i.e., either at the beginning of a line or just before a letter, digit, or underline and after a character which is not one of these.
\>	Forces a match to occur only at the end of a variable or word, i.e., either the end of the line or before a character which is neither a letter, a digit, nor the underline character.
[*string*]	Matches any single character in the class defined by string. Most characters in string define themselves. A pair of characters separated by - in string defines the set of characters collating between the specified lower and upper bounds, thus [a-z] as a regular expression matches any single lowercase letter.
	If the first character of string is an ^, then the construct matches those characters which it otherwise would not; thus [^**a-z**] matches anything but a lowercase letter (and of course a newline character). To place any of the characters ^, [, or - in string, escape them with a preceding backslash (\).

Combining Regular Expression Primitives

The concatenation of two regular expressions matches the leftmost and then longest string which can be divided, with the first piece matching the first regular expression and the second piece matching the second. Any of the (single-character matching) regular expressions mentioned above may be followed by the character * to form a regular expression which matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows.

The character ~ may be used in a regular expression and matches the text which defined the replacement part of the last substitute command. A regular expression may be enclosed between the sequences **\(** and **\)** with side effects in the substitute replacement patterns.

Substitute Replacement Patterns

The basic metacharacters for the replacement pattern are **&** and ~; these are given as **\&** and \~ when nomagic is set. Each instance of **&** is replaced by the characters which the regular expression matched. The ~ metacharacter stands (in the replacement pattern) for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by the escape character (\). The \n sequence is replaced by the text matched by the nth regular subexpression enclosed between **\(** and **\)**. When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of **\(** starting from the left. The sequences **\u** and **\l** cause the immediately following character in the replacement to be converted to uppercase or lowercase, respectively, if this character is a letter. The sequences **\U** and **\L** turn such conversion on either until **\E** or **\e** is encountered or until the end of the replacement pattern.

Option Descriptions

The set options for the **ex** editor are covered in Section 12. They are included in the subsection "Options, Set, and Editor Startup Files."

Limitations

Editor limits that the user is likely to encounter are as follows: 1024 characters per line, 256 characters per global command list, 128 characters per file name, 128 characters in the previous inserted and deleted text in open or visual, 100 characters in a shell escape command, 63 characters in a string valued option, and 30 characters in a tag name, and a limit of 250,000 lines if the file is silently enforced.

The visual implementation limits to 32 the number of macros defined with map and the total number of characters in macros to be less than 512.

Stream Editor (sed)

The stream editor (**sed**) is a noninteractive text editor that runs on the CENTIX operating system. It is a lineal descendant of the text editor (**ed**). Changes between **ed** and **sed** exist because of differences between interactive and noninteractive operations. The **sed** software is especially useful in the following cases:

- Editing files too large for comfortable interactive editing.
- Editing any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
- Performing multiple global editing functions efficiently in one pass through the input file.

The effective size of a file that can be edited is limited only by the requirement that input and output files fit simultaneously into available secondary storage. This is because only a few lines of the input file reside in memory at one time and no temporary files are used.

Complicated editing scripts can be created separately and given to the **sed** program as a command file. For complex edits, this saves considerable typing and attendant errors. The **sed** program running from a command file is more efficient than an interactive editor even if that editor can be driven by a prewritten script.

Principal loss of functions, compared to an interactive editor, are include the following drawbacks:

- Lack of relative addressing (because of the line-at-a- time operation).
- Lack of immediate verification that a command has done what was intended.

Overall Operation

The **sed** program by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by arguments on the command line.

Command Line

The general format of an editing command is

sed [**-n**] [**-e** *script*] [**-f** s*filename*] [*filenames*]

The general format for the *script* entry is

[*address1,address2*] *function* [*arguments*]

- One or both addresses may be omitted.
- The function must be present.
- Any number of blanks or tabs may separate addresses from the function.
- Tab characters and spaces at the beginning of lines are ignored.
- Arguments may be required or optional according to the function given. Three arguments are recognized on the **sed** command line:

-n	Copy only those lines specified by **p** (print) functions or **p** arguments after **s** (substitute) functions.
-e	Take the next argument as an editing command.
-f	Take the next argument as a file name; the file should contain editing commands, one to a line.

Order of Application of Editing Commands

All editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file) and before any input file is opened.

- Commands are compiled in the order encountered; generally, the order they will be attempted at execution time.
- Commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the **t** (test substitution) and **b** (branch) flow-of-control commands. When the order of application is changed by these commands, it remains true that the input line to any command is the output of any previously applied command.

Pattern Space

The range of pattern matches is called the pattern space. 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 next command (**n**).

Examples

The examples described in the following subsections use the following standard input text, except where noted:

```
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.
```

The command

```
2q
```

will copy the first two lines of the input and quit. The output will be

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

Selecting Lines for Editing

Input file lines that editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address pair) by grouping commands with braces ({ }).

Line Number Addresses

A line number is a decimal integer. As each line is read from the input, a line number counter is incremented. A line number address matches (selects) the input line causing the internal counter to equal the address line number. The counter runs cumulatively through multiple input files. It is not reset when a new input file is opened. As a special case, the **$** character matches the last line of the last input file.

Context Addresses

A context address is a pattern (a regular expression) enclosed in slashes (/*string*/). Regular expressions recognized by the **sed** program are constructed as follows:

- An ordinary character is a regular expression and matches that character.
- A circumflex (^) at the beginning of a regular expression matches the null character at the beginning of a line.
- A dollar sign (**$**) at the end of a regular expression matches the null character at the end of a line.
- The \n character matches an embedded newline character but not the newline character at the end of the pattern space.
- 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 0) 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. If, however, the first character of the string is a circumflex (^), the regular expression matches any character except the characters in the string and the terminal newline character of the pattern space. The circumflex is the only metacharacter recognized within the square brackets. If] needs to be in the set of square brackets, it should be the first nonmetacharacter. For example:

[ ]...]	Includes]
[^]...]	Does not include]

- A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
- A regular expression between the sequences \(and \) is identical in effect to the unadorned regular expression but has side effects which are described under the **s** command (substitute function) below.
- The expression **d** means the same string of characters matched by an expression enclosed in \(and \) earlier in the same pattern. The **d** is a single digit; the string specified is that beginning with occurrence **d** of \(counting from the left. For example, the following expression matches a line beginning with two repeated occurrences of the same string:

  ```
  ^\(.*\)\1
  ```
- The null regular expression standing alone (for example, //) is equivalent to the last regular expression compiled.
- Special characters (^ $. * [ ] \ /), 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.

Number of Addresses

Commands can have 0, 1, or 2 addresses. The maximum number of allowed addresses is given under each command. It is considered an error when a command has more addresses than the maximum allowed.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines that match that address.

If a command has two addresses, it is applied to the first line that matches the first address and to all subsequent lines until (and including) the first subsequent line which matches the second address. An attempt is made on subsequent lines to again match the first address, and the process is repeated. Two addresses are separated by a comma. The following indicates some command examples applied to the standard input text and the resulting match.

/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 1, 3, and 4 (number = 0)
/\(an\).*\1/	matches line 1

Functions

Functions are named by a single alphabetic character. In the following function summaries, the maximum number of allowable addresses is enclosed in parentheses followed by the single character function name. Possible arguments follow in italics, and a description of each function is given.

Whole Line Oriented Functions

(2)d	The d function 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 function 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.
(2)n	The n function reads the next line from the input replacing the current line, and the current line is written to the output. The list of editing commands is continued following the n command.

(1)**a**\ *text* The **a** function causes the text argument (*text*) 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. The *text* is terminated by the first unhidden newline character not immediately preceded by a backslash.

Once an **a** function is successfully executed, text will be written to the output regardless of what later commands do to the line that triggered it. 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** function does not cause a change in the line number counter.

(1)**i**\ *text* The **i** function causes the text argument (*text*) 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 unhidden newline characternot immediately preceded by a backslash.

Once an **i** function 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** function does not cause a change in the line number counter.

(2)**c**\ *text* The **c** function deletes lines selected by its addresses and replaces them with the lines in the text argument (*text*). 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** function, no further commands are attempted on the corpse. If text is appended after a line by **a** or **r** functions and the line is subsequently changed, the text inserted by the **c** function will be placed before the text of the **a** or **r** functions (the **r** function is described later).

Leading blanks and tabs will disappear, as in **sed** commands, for text put in the output by these functions. To get leading blanks and tabs into the output, the first desired blank or tab is preceded by a backslash. The backslash will not appear in the output. The list of editing commands, for example:

```
n
a\
XXXX
d
```

applied to the standard input produces

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n
i\
XXXX
d
```

or

```
n
c\
XXXX
```

Substitute Function

One important substitute function that changes parts of lines selected by a context search within the line is

(2)s[*pattern*][*replacement*][*flags*]

The **s** function replaces the part of a line selected by *pattern* with *replacement*. It can be read

Substitute for *pattern*, *replacement*

Pattern

The pattern argument (*pattern*) contains a pattern exactly like the patterns in 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 (described below) is invoked.

Replacement

The replacement argument (*replacement*) 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, 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 (*flags*) may contain the following:

g	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 the line if a successful replacement was done. The **p** flag causes the line to be written to the output if and only if a substitution was actually made by the **s** function. If several **s** functions, each followed by a **p** flag, successfully substitute in the same input line, multiple copies of the line will be written to the output — one for each successful substitution.

w *filename* Write the line to a file if a successful replacement was done. The **w** flag causes lines which are actually substituted by the **s** function to be written to a file named by *filename.* If *filename* exists before **sed** is run, it is overwritten; if not, it is created. A single space must separate **w** and *filename.* The possibilities of multiple somewhat different copies of one input line being written are the same as for **p**. A maximum of ten different file names may be mentioned after **w** flags and **w** functions.

Examples

The command

```
    s/to/by/w changes
```

applied to the example input described earlier produces on the output

```
    In Xanadu did Kubla Khan
    A stately pleasure dome decree:
    Where Alph, the sacred river, ran
    Through caverns measureless by man
    Down by a sunless sea.
```

and on the file named changes

```
    Through caverns measureless by man
    Down by a sunless sea.
```

If the no-copy option is in effect, the command

```
    s/[.,;?:]/*P&*/gp
```

produces

```
    A stately pleasure dome decree*P:*
    Where Alph*P,* the sacred river*P,* ran
    Down to a sunless sea*P.*
```

To illustrate the effect of the **g** flag, the command

```
    /X/s/an/AN/p
```

produces (assuming no-copy mode)

```
    In XANadu did Kubla Khan
```

and the command

```
    /X/s/an/AN/gp
```

produces

```
    In XANadu did Kubla KhAN
```

Input/Output Functions

(2)**p**	The **print** function writes addressed lines to the standard output file. They are written at the time the **p** function is encountered regardless of what succeeding editing commands may do to the lines.
(2)**w** *filename*	The **write** function writes addressed lines to the file named by *filename.* If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line regardless of what subsequent editing commands may do to them. Exactly one space must separate the **w** and *filename.* A maximum of ten different files may be mentioned in write functions and **w** flags after **s** functions combined.
(1)**r** *filename*	The **read** function reads the contents of *filename* and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line that matched its address. If **r** and **a** functions are executed on the same line, the text from **a** functions and **r** functions is written to the output in the order that the functions are executed. Exactly one space must separate the **r** and *filename.* If a file mentioned by an **r** function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

Note: *Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in **w** functions or flags. That number is reduced by one if any r functions are present (only one read file is 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

```
/Kubla/r note1
```

produces

```
In Xanadu did Kubla Khan

 Note: Kubla Khan (more properly Kublai
 Khan; 1216-1294) was the grandson and
 most eminent successor of Genghiz
 (Chingiz) Khan and founder of the Mongol
 dynasty in China.

A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Multiple Input Line Functions

Three functions, all spelled with capital letters, deal with pattern spaces containing embedded newline characters. They are intended principally to provide pattern matches across lines in the input.

(2)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.

(2)D Delete first part of the pattern space. Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline 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.

(2)P Print first part of the pattern space. Print up to and including the first newline character in the pattern space.

The **P** and **D** functions are equivalent to their lowercase counterparts if there are no embedded newline characters in the pattern space.

Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)**h** Hold pattern space.

The **h** function copies contents of the pattern space into a hold area destroying previous contents.

(2)**H** Hold pattern space.

The **H** function appends contents of the pattern space to contents of the hold area. Former and new contents are separated by a newline character.

(2)**g** Get contents of hold area.

The **g** function copies contents of the hold area into the pattern space destroying previous contents.

(2)**G** Get contents of hold area.

The **G** function appends contents of the hold area to contents of the pattern space. Former and new contents are separated by a newline character.

(2)**x** Exchange.

The exchange command interchanges contents of the pattern space and the hold area.

The following are examples:

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

when applied to the standard input text 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
```

Flow of Control Functions

These functions do no editing on the input lines but control the application of functions to the lines selected by the address part.

(2)! Don't.

The don't command causes the next command (written on the same line) to be applied to those input lines not selected by the address part.

(2){ Grouping.

The grouping command causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the { or on the next line. The group of commands is terminated by a matching } standing on a line by itself. Groups can be nested.

(0):*label* Place a label.

The label function marks a place in the list of editing commands that may be referred to by **b** and **t** functions. The *label* argument may be any sequence of eight or fewer characters. If two different colon functions have identical labels, a compile time diagnostic will be generated; and no execution attempted.

(2)**b***label* Branch to *label*.

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same *label* was encountered. If no colon function with the same label can be found after all editing commands have been compiled, a compile time diagnostic is produced; and no execution is attempted.

A **b** function 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.

(2)**t***label* Test substitutions.

The **t** function tests whether any successful substitutions have been made on the current input line; if so, it branches to *label*; if not, it does nothing. The flag that indicates a successful substitution has been executed is reset by reading a new input line and executing a **t** function.

Miscellaneous Functions

(1)=	The = function writes to standard output the line number of the line matched by its address.
(1)**q**	The **q** function causes the specified lines to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

Big File Scanner (bfs)

Big file scanner (**bfs**) is similar to **ed** except that it is read-only and processes much larger files. Files can be up to 1024K bytes (the maximum possible size) and 32K lines, with up to 512 characters, including newline, per line (255 for 16-bit machines). **Bfs** is usually more efficient than **ed** for scanning a file since the file is not copied to a buffer. It is most useful for identifying sections of a large file where **csplit (1)** can be used to divide it into more manageable pieces for editing.

Normally, the size of the file being scanned is printed, as is the size of any file written with the **w** command. The optional - suppresses printing of sizes. Input is prompted with * if **P** and a carriage return are typed as in **ed**. Prompting can be turned off again by inputting another **P** and carriage return. Note that messages are given in response to errors if prompting is turned on.

All address expressions described under **ed** are supported. In addition, regular expressions may be surrounded with two symbols besides / and **?**: > indicates downward search without wraparound, and < indicates upward search without wraparound. There is a slight difference in mark names: only the letters **a** through **z** may be used, and all 26 marks are remembered.

The **e, g, v, k, p, q, w,** =, ! and null commands operate as described under **ed** . Commands such as ---, +++-, +++=, **-12,** and **A4p** are accepted. Note that **1,10p** and **1,10** will both print the first ten lines. The **f** command only prints the name of the file being scanned; there is no remembered file name. The **w** command is independent of output diversion, truncation, or crunching (see the **xo, xt,** and **xc** commands, below). The following additional commands are available:

xf *filename*

Further commands are taken from the named *filename*. When an end-of-file is reached, an interrupt signal is received or an error occurs; reading resumes with the file containing the **xf.** **Xf** commands may be nested to a depth of ten.

xn

List the marks currently in use (marks are set by the **k** command).

xo [*filename*]

Further output from the **p** and null commands is diverted to the named *filename*. If *filename* is missing, output is diverted to the standard output. Note that each diversion causes truncation or creation of the file.

: *label*

This positions a *label* in a command file. The *label* is terminated by newline, and blanks between the : and the start of the label are ignored. This command may also be used to insert comments into a command file since labels need not be referenced.

(.,.)**xb**/*regular expression*/*label*

A jump (either upward or downward) is made to *label* if the command succeeds. It fails under any of the following conditions:

- Either address is not between **1** and **$**.
- The second address is less than the first.
- The regular expression does not match at least one line in the specified range, including the first and last lines.

On success, . is set to the line matched and a jump is made to *label*. This command is the only one that does not issue an error message on bad addresses, so it may be used to test whether addresses are bad before other commands are executed. Note that the command

```
    xb/^/ label
```

is an unconditional jump. The **xb** command is allowed only if it is read from someplace other than a terminal. If it is read from a pipe, only a downward jump is possible.

xt *number*

Output from the **p** and null commands is truncated to at most *number* characters. The initial number is 255.

xv[*digit*][*spaces*][*value*]

The variable name is the specified digit following the **xv**. **xv5100** or **xv5 100** both assign the value 100 to the variable 5. **Xv61,100p** assigns the value 1,100p to the variable 6. To reference a variable, put a **%** in front of the variable name. For example, using the above assignments for variables 5 and 6:

```
1,%5p
1,%5
%6
```

will all print the first 100 lines.

```
g/%5/p
```

would globally search for the characters 100 and print each line containing a match. To escape the special meaning of **%**, a \ must precede it.

```
g/".*[cds]/p
```

could be used to match and list lines containing *printf* of characters, decimal integers, or strings. Another feature of the **xv** command is that the first line of output from a CENTIX system command can be stored into a variable. The only requirement is that the first character of *value* be an !. For example:

```
.w junk
xv5!cat junk
!rm junk
!echo "%5"
xv6!expr %6 + 1
```

would put the current line into variable 5, print it, and increment the variable 6 by one. To escape the special meaning of ! as the first character of *value*, precede it with a \.

```
xv7date
```

stores the value !date into variable 7.

xbz *label*

xbn *label*

These two commands will test the last saved return code from the execution of a CENTIX system command (*!command*) or nonzero value, respectively, to the specified label. The two examples below both search for the next five lines containing the string size.

```
xv55
: l
/size/
xv5!expr %5 - 1
!if 0%5 != 0 exit 2
xbn l
xv45
: l
/size/
xv4!expr %4 - 1
!if 0%4 = 0 exit 2
xbz l
```

xc [*switch*]

If *switch* is 1, output from the **p** and null commands is crunched; if *switch* is 0 it is not. Without an argument, **xc** reverses *switch*. Initially *switch* is set for no crunching. Crunched output has strings of tabs and blanks reduced to one blank and blank lines suppressed.

Editing BTOS Files with vi and ed

If your CENTIX system is also running BTOS, the **ofvi** and **ofed** commands allow you to edit BTOS files from your PT 1500. These commands call up the **vi** and **ed** editors, allowing you to edit BTOS files in the same manner as you would any CENTIX file.

The **ofvi** command calls up the **vi** editor and **ofed** calls up the **ed** editor. Once the BTOS file has been accessed, both editors behave exactly as though you were editing a CENTIX file.

To use either **vi** or **ed** to access and edit a BTOS file, perform the procedure outlined below:

1 Log onto the PT 1500 as **root**. (You must have super-user status in order to execute these commands.)

  The pound prompt (#) appears.

2 Enter **ofvi** (or **ofed**, depending on which editor you prefer to use) followed by the full path name of the BTOS file you want to edit. Place single quotes around the BTOS path name.

3 Press the RETURN key and the selected BTOS file appears on your screen.

For example, to use the **vi** editor to edit the file [sys]<sys>ConfigUFS.sys, you would log in as **root** and then make the following entry:

```
# ofvi '[sys]<sys>ConfigUFS.sys'
```

Then press the RETURN key.

If access to the BTOS file is password protected, append a caret (^) to the end of the file name, followed by the required password.

Index

Title: ____________________

Form Number: ____________________ Date: __________

Burroughs Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information.

Please check type of suggestion: ☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments: ____________________

____

____

____

Name ____________________

Title ____________________

Company ____________________

Address ____________________
Street City State Zip

Telephone Number (   ) ____________________
Area Code

Title: ____________________

Form Number: ____________________ Date: __________

Burroughs Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information.

Please check type of suggestion: ☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments: ____________________

____

____

____

Name ____________________

Title ____________________

Company ____________________

Address ____________________
Street City State Zip

Telephone Number (   ) ____________________
Area Code

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 817 DETROIT, MI 48232

POSTAGE WILL BE PAID BY ADDRESSEE

Burroughs Corporation
Production Services – East
209 W. Lancaster Avenue
Paoli, Pa 19301 USA

ATTN: Corporate Product Information

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 817 DETROIT, MI 48232

POSTAGE WILL BE PAID BY ADDRESSEE

Burroughs Corporation
Production Services – East
209 W. Lancaster Avenue
Paoli, Pa 19301 USA

ATTN: Corporate Product Information