**Replace this**

**page with the**

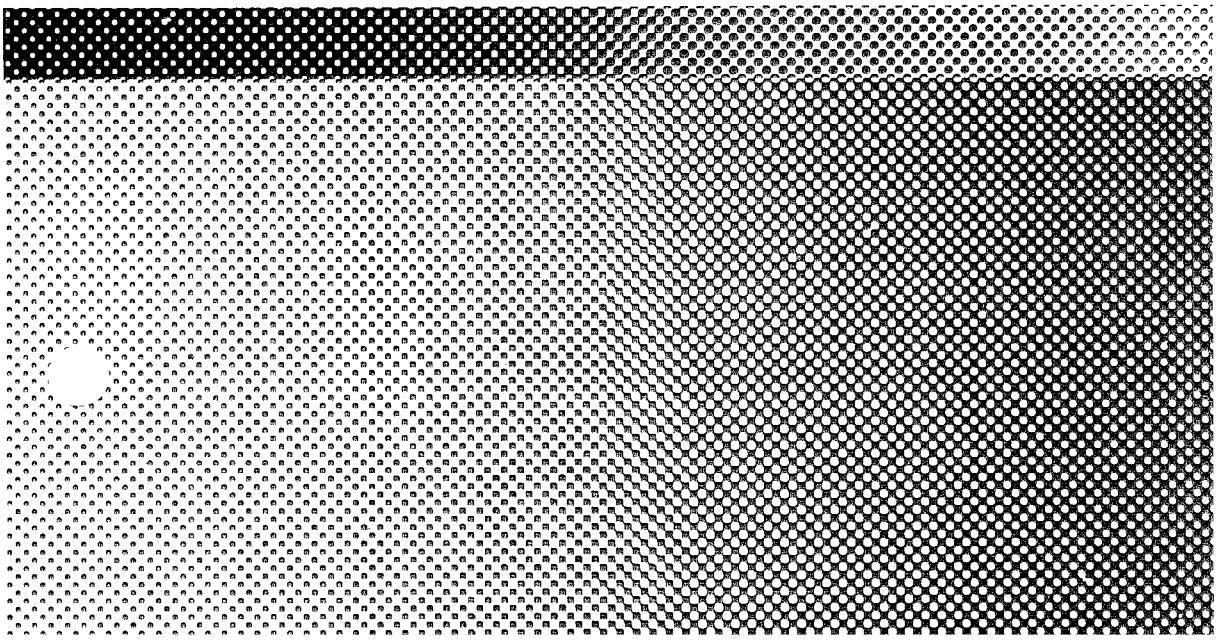
*EDITING*

**tab separator.**

AT&T

# AT&T 3B2 Computer

## UNIX™ System V Release 2.0 Editing Utilities Guide



# CONTENTS

# Chapter 1

# INTRODUCTION

## GENERAL

This guide describes the command format and use of the Editing Utilities. The commands and procedures described in this guide are for use by all users.

The **UNIX*** System contains a file system that is used to store user information. Changing files by adding or deleting information can only be done using UNIX System Editor Commands. The editing utilities give the user an easy way to create, read, and change information in these files.

The edit, ex, and vi editors are based on a consistent set of text editor commands. These commands serve as the fundamental building blocks for increasing text editing proficiency.

---

* Trademark of AT&T

The editing utilities allows the user to do two types of editing:

- **Basic editing** allows the casual user to use basic commands to do text editing.
- **Visual editing** allows the user to view several lines of the file at a time and use screen oriented display editing based on basic editor commands.

The editing utilities consists of three text editors designed to meet the needs of the novice user, while allowing the experienced user to use more complex and powerful editing tools. These editors are actually three versions of the ex editor.

The ex editor is an interactive editor that normally accesses only one line of the file at a time. Many of the ex commands are similar to the ed editor commands. The advantage of using the ex editor is the large amount of options available in it.

The edit editor is the simplified version of ex editor and is normally used by novice users. Messages displayed on the screen after an invalid command are more descriptive than with ex or vi. Edit contains fewer commands and most beginners should pick it up quickly. All commands that execute in the edit editor will also execute in the ex editor.

The vi editor is actually the visual mode of editing within the ex editor. Vi is the most complex of the three editors, because there are so many commands that do the same function. However, it is the easiest to use once you understand the basic movement and editing commands. With the vi editor, you can view several lines of the file at one time, and you can move the cursor to any character in the file. Most ex commands can be invoked separately from vi by first entering a ":" and then the ex command. To execute the command, depress the carriage return. Experienced users often mix their use of ex command mode and vi command mode to speed the work they are doing.

## RESTRICTIONS

The limits of the editors are as follows:

- 1024 characters per line
- 256 characters per global command list
- 128 characters per file name
- 100 characters per shell escape command
- 63 characters in a string valued option
- 30 characters in a tag name
- 128 characters in the previous inserted or deleted text in (open) or (visual) mode
- 250000 lines in a file.

If you try to use these editors on a file and you receive a message stating that the file is too large, you can either split the file into smaller files or use a different editor. To split the file, you can use the **split** or **csplit** commands contained in the *AT&T 3B2 Computer Directory and File Management Utilities*. If you want to use another editor, you can use the **bfs** editor (big file scanner) contained in the *AT&T 3B2 Computer Directory and File Management Utilities* or the **sed** (stream) editor contained in the *Essential Utilities*.

## SPECIAL PURPOSE KEYS

There are several special purpose keys that are used by the vi editor. These keys are important and will be used throughout the document. Their descriptions are as follows:

ESCAPE This key is sometimes labeled *<ESC>* or *<ALT>*. It is normally located in the upper left corner of your keyboard. When you are in the editor, depressing the *<ESC>* key causes the editor to ring the bell indicating that it is in an inactive state. On smart terminals where it is possible, the editor will quietly flash the screen rather than ring the bell. Partially formed commands are canceled with the *<ESC>* key. When you insert text in the file, text insertion is ended with the *<ESC>* key. This is a harmless key to use, so you can depress it whenever you are not certain what state the editor is in.

CR The *<CR>* key refers to the RETURN key and is used to start execution of certain commands. It is normally located on the right side of the keyboard.

DELETE This key is sometimes labeled *<DEL>*, *<RUBOUT>*, or *<BREAK>*. It 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 if you do not know or like what is going on.

CONTROL This key is often labeled *<CTRL>*. It is used with other keys to do various functions. It will be represented in this document by the *<CTRL>* symbol. The associated key will be represented by an uppercase letter. To execute a control function, both keys must be depressed at the same time. An example of this will be represented as follows:

*<CTRL d>*

The function illustrated will cause the screen to scroll down when in the vi editor.

## HOW TO INTERPRET COMMANDS

The following conventions are used to show your terminal input and the system output in screens and command lines:

This style of type is used to show system generated responses displayed on your screen.

**This style of bold type is used to show inputs entered from your keyboard that are displayed on your screen.**

These bracket symbols, < > identify inputs from the keyboard that are not displayed on your screen, such as: <*CR*> carriage return, <*CTRL d*> control d, <*ESC g*> escape g, passwords, and tabs.

*This style of italic type is used for notes that provide you with additional information.*

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

## GUIDE ORGANIZATION

This guide is structured so you can easily find desired information without having to read the entire text. The remainder of this document is organized as follows:

- Chapter 2, "EDIT EDITOR," provides instructions on how to use the edit editor.
- Chapter 3, "EX EDITOR," provides instructions on how to use the ex editor.
- Chapter 4, "VI EDITOR," provides instructions on how to use the visual (vi) editor.

# Chapter 2

# EDIT EDITOR

PAGE

# Chapter 2

# EDIT EDITOR

## INTRODUCTION

This chapter describes the edit editor used on the 3B2 Computer. Edit is a simplified version of the ex editor, and it is recommended for new or casual users. Messages displayed on the screen after an invalid command are more descriptive than with the other editors.

When using the edit editor, all commands must be entered on a command line. The command line is identified by a colon ":" on a line by itself. Commands entered on the command line can affect the line you are on in the file (current line), a specified set of lines, or the entire file.

Most edit editor command names are English words that can be abbreviated. When an abbreviation conflict is possible, the more commonly used command has the shorter abbreviation. For example, since **substitute** is abbreviated by **s**, **set** is abbreviated by **se**.

The edit editor does not directly change the file being edited. Instead, it works on a copy of the file stored in a temporary memory location called the buffer. The edited file is not changed until you write the changes from the buffer to the edited file.

This editor description assumes that you know how to log on to the computer. If you do not, refer to the *AT&T 3B2 Computer Owner/Operator Manual.*

For additional information on the edit editor, refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages.

## CURRENT LINE DEFINITION

The term "current line" is referred to throughout this chapter. The current line is the line in the file you are now on. Each time you move to a different line in the file, that line becomes the current line. Whenever a command is given, the current line is used as a reference point. Any command that is not directed at any specific line is executed against the current line.

## GETTING STARTED

The edit editor can be used to create a new file or to change an existing file. To execute edit you must be logged onto the computer. After the **$** or # prompt is displayed, you can begin working with the edit editor.

## Creating a New File

To create a new file, you will need to type **edit** followed by a space and then the name of the file you wish to create. Execute the command by depressing the carriage return <*CR*>. For example:

```
$ edit filename<CR>
" filename" [New file]
:
```

If you did not enter the command correctly, you will receive a usage message indicating an incorrect command syntax was used. You will need to re-enter the command correctly.

If you entered the **edit** command without a filename, the editor will still create a new file. However, when you decide to write the file into memory you will be prompted for a filename. See "Writing the Buffer Into the File."

When the edit command is executed, a colon ":" is displayed. The colon identifies the command line and indicates that the edit editor is ready to accept your input commands.

## Entering Text

The edit editor commands have two forms: a word that describes what the command does and an abbreviation of the word. You can use either form. Many beginners find the full command name easier to remember, but after some practice use the abbreviation. The command to input text is **append**, that may be abbreviated **a**. Enter **append** after the colon on the command line and then depress the carriage return.

:**append**<*CR*> or :**a**<*CR*>

Edit is now in the *text input mode* (append mode). The colon is no longer displayed on the command line, and this is your signal that you may begin entering lines of text. Anything that you type on your terminal, except a period on a line by itself, is entered into the buffer. If the message:

```
Not an editor command
```

is displayed, check to see what you entered incorrectly and then enter the command again.

> ***Note:*** The computer considers a blank space to be a character. Be careful not to input blanks into lines of text unless you mean for them to be there.

## Leaving the Input Mode

To leave the input mode, simply enter a period "." on a line by itself and depress the carriage return. This is the signal that you want to stop inputting text. After receiving a period on a line by itself, edit will re-enter the command mode and display the command line prompt ":".

The text just entered is now stored only in the buffer. If you wish, you can make changes to the text. Making changes is discussed throughout the remainder of this chapter.

## Writing the Buffer Into the File

The buffer is only temporary storage for the file. Now that you have entered text in the buffer, you need to write the buffer to the file. This is the only way to save new text from one editing session to another. To write the contents of the buffer to the file, use the **write** command (abbreviated **w**).

:**write**<*CR*> or :**w**<*CR*>

Edit will then copy the buffer into the file. If the file does not yet exist, a new file will be created and a message will be given indicating that it is a

new file. The newly created file will be given the name specified when you entered the editor, "filename". To confirm that the file has been successfully written, the editor will repeat the filename and give the number of lines and the total number of characters in the file. The buffer remains unchanged, so you can make further changes if you want to.

Edit must have a filename to use before it can write a file. Therefore, if you did not specify the name of the file when you began the editing session, edit will issue the message:

```
No current filename
```

when you give the **write** command. If this happens, simply re-enter the **write** command and specify the filename. Here you would enter:

**:write filename**<*CR*> or **:w filename**<*CR*>

This will write the buffer to a file named "filename".

## Quitting the Editor

When you have finished editing the file and you are ready to return to the UNIX System, enter the **quit** command (abbreviated **q**).

**:quit**<*CR*> or **:q**<*CR*>

This returns you to the UNIX System unless you forget to write the buffer to the file. The system will issue a message reminding you to write the file. A quick way to **write** and **quit** the edit editor is with the single command:

**:wq**<*CR*>

If you do not want to save the changes, enter the command:

**:q!**<*CR*>

This will quit edit and leave the file unchanged from the last write command.

## Editing an Existing File

To edit the contents of an existing file named "file1," you begin by issuing the command:

```
$ edit file1<CR>
"file1" 150 lines, 4285 characters
:
```

This places a copy of the file in a buffer, and displays how many lines and characters are in the file. A colon ":" will then be displayed at the command line.

> ***Note:*** If you do not give a filename, edit will create a new file instead of editing the file you want.

After the file description and the colon ":" are displayed, enter a **1** on the command line followed by a carriage return. This will make the first line in the file the current line. The editing process is described throughout the remainder of this chapter.

The procedure for saving changes to the buffer is described in "Writing the Buffer Into the File." The procedure for quitting the editor is described in "Quitting the Editor."

## DISPLAYING LINES IN THE FILE

When editing a file, you should always display the current line before making changes. This is important since most commands are executed on the current line. After making any changes, display the lines again to make sure you are happy with the changes. If you do not like the changes, you can use the **undo** command described in "RECOVERING LOST TEXT."

To display a line, all you need to do is depress the carriage return. This will display the current line in the editor. Each time you depress the carriage return, the next line is displayed, and it becomes the current line.

If you want to display the entire contents of the buffer, enter the command:

**:1,$print**<*CR*> or **:1,$p**<*CR*>

The "1" stands for line 1 of the buffer, the "$" is a special symbol designating the last line of the buffer, and the "p" is the command to print from line 1 to the end of the buffer. After displaying the buffer, the last line becomes the current line.

Occasionally, characters that do not appear on your terminal screen are contained in a line of text. These characters are normally called "control characters" because the control key was depressed when they were entered. To display all the characters in a line, including control characters, you can use the **list** command instead of the **print** command. For example:

**:5,201list**<*CR*> or **:5,201**<*CR*>

will display any character contained in that line regardless of what type it is. The **list** command executes exactly the same way the **print** command does.

# MOVING AROUND IN THE FILE

## Basic Movement Commands

Edit accepts "-" and "+" as movement commands. As you would expect, - moves the current line backwards and + moves the current line forwards. With these commands you can move to adjacent lines in the buffer.

You can move more than one line at a time by using numbers with the + and - commands. For example:

:-5<*CR*>

moves the current line backwards 5 lines from its current position and displays the line. Likewise,

:+25<*CR*>

moves the current line forwards 25 lines from its current position and displays the line. This makes it much easier to move to the line you want to work on. Another useful command is:

:$<*CR*>

that moves the current line to the last line in the buffer and displays the line.

Each line in the file has a line number associated with it, although they are not displayed. Edit allows you to move across large areas of the buffer by entering the line number and a carriage return. For example:

:43<*CR*>

makes 43 the current line and displays the line.

## Forward and Backward Search Commands

If you are not sure where a line you want to change is, but you know an exact pattern of characters on the line, you can search for that pattern. **The pattern must be on one line.** The command line interprets the character "/" as meaning "search for this pattern." The *search* command "/" searches from your present position forward through the buffer for the first occurrence of the pattern. For example, if you know the pattern "learning to use edit" is somewhere in the buffer, you can find it by executing the command:

```
:/learning to use edit/p<CR>
```

This will make the line containing this pattern the current line and display the line. If you leave the "p" off the command, edit will still search for the pattern and make it the current line, but will not display the line. Always include the **p** as part of the search command. The pattern may be used more than once in the buffer.

If you execute a search, but edit cannot find the pattern, the message:

```
Pattern not found
```

will be displayed. This means the pattern you searched for is not in the buffer and the current line does not change. Check to see if you correctly entered the search command or if it included any characters with special meanings. (See Special Search Characters.)

The character "**?**" also executes a search when used on the command line. It works the same as the "/" search character, except that it searches backwards from your present position in the buffer.

## Repeating Searches

When searching for a pattern, the first occurrence is not the one that you are actually looking for. You could repeat the search command, but there is a much easier way. The editor remembers the last search pattern entered. If you enter the command:

:**//**<*CR*>

a forward search will look for the remembered pattern. The backwards search command **??** will also repeat searches. The repeated search does not have to be the same type as the original search.

## Global Searches

The edit editor also allows you to do global searches on the file. A global search is used to find all the occurrences of a specified pattern in a file. This type of search is useful when scanning for a pattern that occurs in several places. The two types of global searches that can be executed use the **g** and **v** commands.

The global search that uses the **g** command locates all the lines that contain a specified pattern. An example would be:

:**g/sample pattern/p**<*CR*>

This will search for and display all lines containing the words "sample pattern". The current line will be the last line displayed.

The global search that uses the **v** command locates all lines that **do not** contain a specified pattern. An example would be:

:**v/sample pattern/p**<*CR*>

This will search for and display all lines that do not contain the words "sample pattern". The current line will be the last line displayed.

## Special Search Characters

Several characters have special meaning when used in specifying searches. These characters will work with all types of searches. They can be used to: match repetitive strings of characters, turn off special meanings of characters, or denote the placement of characters in the line. These characters and their use are explained below:

. The period matches any single character except the newline (carriage return) character. For example, if a line in your file contains the words "edit editor", or a pattern with any other character between "edit edit" and "r", you could find the line by entering the command:

:/f(BB/edit edit.r/<*CR*>

* The asterisk matches any repeated characters except the first ., \, [, or ~ in that group. For example, if a line in your file contains the pattern "the xxxx editor", you could search for the line by entering the command:

**:/the x* editor/**<*CR*>

[] Brackets are used to enclose a variable set of characters. For example, if a line in your file contains the patterns"file2", "file3", and "file4", you could search for the first occurrence of these patterns by entering the command:

**:/file[2-4]/**<*CR*>

**$** The dollar sign is interpreted by the editor to mean "end of the line". It is used to identify patterns that occur at the end of a line. For example, if a line in your file ends in the pattern "last character" you could find the line by entering the command:

**:/last character$/**<*CR*>

^ The circumflex (caret) works like "**$**" except it looks for the pattern at the beginning of the line. For example, if a line in your

file begins with the pattern "First character" and you could find the line by entering the command:

:/^First character/<*CR*>

\ The backslash is used to cancel the meaning of the special characters. It should be placed immediately before the character it is to nullify. For example, if a line in your file contains the pattern "This is a $" you could search for it by entering the command:

:/This is a \$/<*CR*>

The character $ will be searched for instead of interpreting it as meaning "end of the line."

To search for the characters ., *, \, [, ], $, or ^, you must precede the characters with a backslash. You can also combine these special characters in one search command. For example, .* can be used to search for any string of characters.

# MAKING CORRECTIONS TO THE FILE

There are several edit commands you can use to make corrections to a file. These commands are: append, input, delete, substitute, change, move, and copy.

## Appending Text

The **append** command (abbreviated **a**) is used to input text in the buffer after the current line. It places edit in the *text input mode*. While in this mode, the colon prompt on the command line is not displayed. Anything you type, except a period on a line by itself, will be entered on lines of text in the buffer. To leave the *text input mode*, simply enter a period "." on a line by itself and depress the carriage return. Edit will then return to the command mode and display the command line prompt ":".

As previously discussed in "GETTING STARTED," the **append** command can be used to input text when the buffer is empty. The **append** command can also be used to input text anywhere in an existing file. The following steps outline how to append text to the current line:

1. Move to the place in the buffer where you want to append text. This can be done using movement commands or a search command. The line you select becomes the current line.

2. Enter the command:

   :**append**<*CR*> or :**a**<*CR*>

   The colon prompt will no longer be displayed on the command line.

3. Enter any text you like using as many lines as you like.

4. To leave the *text input mode* and return to the command mode, enter a period "." on a line by itself and depress the carriage return.

5. The command line prompt ":" will reappear. This indicates that you may enter another edit command.

## Inserting Text

The **insert** command (abbreviated **i**) works similarly to the **append** command. The only difference is that text is inserted before instead of after the current line. To insert text in the buffer, enter:

:**insert**<*CR*> or :**i**<*CR*>

on the command line. You may now begin inserting text. To return to the command mode, simply enter a period "." on a line by itself and depress the carriage return. The command prompt will be displayed on the screen.

## Changing Text

There may be instances when you want to delete one or more lines and insert new text in their place. This can be done easily with the **change** command (abbreviated **c**). The **change** command instructs edit to delete specified lines and then switch to *text input mode* to accept text to replace the lines. The number of lines you insert does not have to match the number deleted. For example, if you want to change the current line, enter:

:**change**<*CR*> or :**c**<*CR*>

The colon prompt will no longer be displayed. You may begin inserting as many lines of text as you want. To return to the command mode, enter a period "." on a line by itself and depress the carriage return.

If you want to replace lines 25 through 34 with some new text, you would enter:

:**25,34c**<*CR*>

Edit will respond with:

```
10 lines changed
```

The colon prompt will no longer be displayed. The procedure for entering text and for returning to the command mode is the same as for changing

one line. By default, if five or fewer lines are changed, edit will not display the number of lines being changed. (See *report* option given in Chapter 4.)

## Deleting Text

The delete command (abbreviated **d**) is an easy command to execute. This command can also be disastrous if you are not careful when using it. To delete the current line, all you have to do is enter:

**:delete**<*CR*> or **:d**<*CR*>

This will delete the line and display the next line which becomes the current line.

> ***Note:*** You can use the **undo** command "**u**" to retrieve deleted lines as long as you have not executed any other commands that changed the buffer. (See "Recovering Lost Text.")

If you know the line number of a line you want to delete, you can enter the line number followed by **delete** or **d**. For example:

**:15d**<*CR*>

will delete line 15. You can also delete a range of lines by using commands such as **2,3d** to delete lines 2 and 3, or **2,8d** to delete lines 2 through 8.

When one or more lines are deleted, the numbers of all following lines are changed. When deleting different groups of lines from a file, it is easier to start with the higher line numbers and work toward the lower line numbers.

If you do not know the line number, you can search for the line and then delete it. Searching for text is discussed in "Forward and Backward Search Commands."

## Substituting Text

To change any characters on an existing line without replacing the whole line, you can use the **substitute** command (abbreviated **s**). The substitute command searches for a specified pattern and then changes the pattern accordingly. The substitute command normally executes on the current line.

> ***Note:*** The **global** option can be used with the substitute command, but you must be careful. (See "Global Substitutes.")

Using the substitute command can sometimes be confusing to a novice user. However, if you think about the parts of the command, it is really easy. The format of the command is:

**:s/***old-pattern***/***new-pattern***/p**

The "s" is the substitute command. The "/old-pattern/" tells edit to search the current line for the pattern. The "new-pattern/" tells edit what to substitute for "old-pattern" and "p" tells edit to display the new form of the current line. For example, if the current line is "Substituting is very confusing." and we want to change it to "Substituting is very easy.", we would use the command:

**:s/confusing/easy/p**<*CR*>

If you want to delete the word "very" from the new sentence, you could use the substitute command and not put a pattern where the new pattern should be.

**:s/very //p**<*CR*>

Your new sentence would be "Substituting is easy." Notice that a blank space was also removed because edit considers it a character.

## Special Substitution Characters

All the special search characters given in "Special Search Characters" are also special characters in the search portion of substitution commands. However, there are two characters that have special meaning when used in the replacement portion of substitute commands. These characters are **&** and ~.

**&** The ampersand (**&**) character is used to save you from having to repeat the search portion of the substitute command when you are only adding characters. For example, if a line in your file contains the pattern "The game is tonight" and you wanted to change it to "The game is tonight at eight" you could use the following substitute command:

```
:s/The game is tonight/& at eight/p<CR>
```

**~** The tilde (~) character works similar to the ampersand (**&**) character, except that it also repeats previous substitution commands.

To turn off the special meaning of the **&** and the ~ in the substitution command, it must be preceded by a backslash (\). These special characters will work with all types of substitution commands.

## Global Substitutes

A global substitute is similar to a regular substitute, except that instead of only working on the current line it works on every line in the buffer. Before trying to understand global substitutes, be sure you understand regular substitutes. (See "Substituting Text.")

You must be careful when using global substitutes. There may be an occasion when you want to use a global substitute, but the pattern you want to search for may not be unique. If you think a line you want left alone might change, first do a global search and display all the lines. You may be able to find a pattern that is unique only to what you want changed. The format of a global substitute is as follows:

**:g/***old-pattern***/s/***old-pattern***/***new-pattern***/gp**

In this example, the "**g**/*old-pattern*/" instructs edit to search for every occurrence of "old-pattern". The "**s**/*old-pattern*/*new-pattern*/" instructs edit to substitute "*new-pattern*" for every occurrence of "*old-pattern*". The "**g**" after the substitute command instructs edit to execute the substitution for every occurrence on each line if "*old-pattern*" is on a line more than one time. The "**p**" tells edit to display all the lines where substitutions were made.

> ***Note:*** The "**g**" at the end of the command should be omitted if you only want the first occurrence of the pattern on each line to change.

When using a global substitute command where the pattern you search for is the same as the pattern you want to change, you can use an abbreviated version of the command. For example, the command:

**:g/***old-pattern***/s//***new-pattern***/gp**

will execute the same as the previous example. This saves you from having to input the pattern (old-pattern) in twice.

Edit also allows you to execute a global substitute within a range of lines. For example:

**:35,75g/***old-pattern/s//new-pattern/gp*<*CR*>

would only do the substitutions from line 35 to line 75. All other lines would not be affected. This option allows you a much greater flexibility when using global substitutes.

If you decide you do not like what happened when you used the global substitute you have two choices. You can either try the **undo** command or you can quit the editor without writing the buffer into the file. (See "RECOVERING LOST TEXT.")

If you are not sure whether you want to keep the changes, you can write the buffer to a new file, and then either use the **undo** command or quit without writing. This way you can review both files before deciding which one to keep. (See "Writing the Buffer to Another File.")

## Copying Text

Edit allows you to create a copy of specified lines in the buffer and insert them where you want by using the **copy** command. The original lines will remain unchanged. The **copy** command has the same format as the **move** command. For example:

**:14,19copy**<*CR*> or **:14,19co**<*CR*>

would create a copy of lines 14 through 19 and place it at the end of the buffer. The original lines 14 through 19 will stay the same. When the command has finished executing, the lines are automatically renumbered.

> ***Note:*** The abbreviation for the copy command is **co**. The **c** command is to change lines of text.

## Moving Text

Edit allows you to move lines of text from one location to another in the buffer by using the **move** command (abbreviated **m**). You are allowed to move as many lines as you want. For example,

**:2m15**<*CR*>

would move line 2 to the position after line 15, and then renumber the lines. If you wanted to move a block of text, you could use the command:

**:2,20m25**<*CR*>

This would move lines 2 through 20 to the position after line 25.

When using the move command, you can specify the end of the buffer by using the **$** character instead of the line number. This is often much easier than looking to see what is the last line number. Two examples of using the **$** in a move command are:

**:15,$m10**<*CR*> and **:1,20m$**<*CR*>

The first example would move lines 15 through the end of the buffer to the position after line 10.

The second example would move lines 1 through 20 to the end of the buffer.

# FILE MANIPULATION

## Writing the Buffer to Another File

The **write** command (abbreviated **w**) allows you to write all or part of the buffer to a new file. This allows you to keep copies of the buffer in various states of change. To write the whole buffer to another file, simply use the write command and the name of the file. For example:

**:write filename**<*CR*> or **:w filename**<*CR*>

Be careful when naming the file. If you use an existing filename, the editor will display the message:

```
"filename" File exists - use "w! filename" to overwrite
```

When this occurs, you can either use a different filename, or use the **w!** command to overwrite the file. If you overwrite the file, the information being overwritten is no longer accessible.

If you only want to write part of the buffer to another file, you must specify the beginning and ending lines you want to write. For example:

**:85,$w save**<*CR*>

will write lines 85 through the end of the buffer to the file named *save*. The write command does not change the buffer.

## Reading Another File Into the Buffer

The **read** command (abbreviated **r**) allows you to input the contents of another file into the buffer without destroying the text already there. To use the read command, first move to the line where you want the file appended. Then enter the read command using the following format:

**:read filename**<*CR*> or **:r filename**<*CR*>

Edit will append a copy of the file after the current line, and issue a message stating the name of the file, the number of lines, and the number of characters that were inserted.

## Obtaining Information About the Buffer

Edit maintains a record of the current information about the buffer. To access this information, enter the **file** command (abbreviated **f**). Edit displays the filename, your current position, and the number of lines in the buffer. If the contents of the buffer have been changed since the last time the file was written, the editor will tell you that the file has been modified. It also displays what per cent of the way you are through the buffer. For example, enter the command:

:**f**<*CR*>

The computer will respond with a message such as:

```
"filename" [Modified] line 15 of 75 --20%--
```

***Note:*** After you save the changes by writing the buffer to the file, the buffer will no longer be considered modified.

## ISSUING UNIX SYSTEM COMMANDS

Edit allows you to execute a single UNIX System command by entering a command of the form:

:**!cmd**<*CR*>

where "cmd" represents the command you want to execute. The system will then execute the command. When finished, edit displays an **!** and then reissues the command line prompt ":". You can then continue editing or enter another UNIX System command.

If you need to execute more than one UNIX System command, enter the command:

:**sh**<*CR*>

When you are finished executing UNIX System commands, enter <CTRL **d**>. The editor will then display the message:

```
[Hit return to continue]
```

After depressing the carriage return, the editor will display the command line prompt.

***Caution: Be sure to write the buffer into the file before escaping to the UNIX System. The editor will normally save the buffer, but it will issue a message to remind you.***

# RECOVERING LOST TEXT

## Undoing the Last Command

The **undo** command (abbreviated **u**) is able to reverse the effects of the last command executed that changed the buffer. This enables you to restore the buffer after making an editing mistake. To execute the undo command enter:

:**undo**<*CR*> or :**u**<*CR*>

The undo command only works on commands such as append, insert, delete, change, move, copy, and substitute. You can also undo an undo if you decide to keep the change. Commands that do not affect the buffer such as: write, edit, and print cannot be undone.

## Recovering Lost Files

If the system crashes, you can recover the contents of the buffer by using the **recover** command. **The recover command cannot be abbreviated.** You will normally receive mail the next time you log in, giving you the name of the file that was saved for you. You should then change to the directory containing the file being edited when the system crashed. Then access the file by entering:

:**edit filename**<*CR*>

replacing "filename" with the name of the lost file. Once in the editor, enter:

:**recover filename**<*CR*>

Recover is sometimes unable to save the entire contents of the buffer, so always check the contents of the saved buffer before writing it back to the original file.

If something goes wrong with the editor when you are using it, do not leave the editor. You may be able to save your work by using the **preserve** command (abbreviated **pre**). This saves the buffer as if the system had crashed.

If you are writing the buffer into the file and you get the message:

```
Quota exceeded
```

you have tried to use more disk space than you are allotted. When this happens, it is likely that only part of the buffer was written into the file. When this happens you should escape to the UNIX System using the **sh** command and remove some files you do not need. Then, try writing the file again. If this is not possible, enter the command:

:**preserve**<*CR*>

and then get help from the person who is administrating the system. **Do not quit the editor or your buffer will be lost.**

After using the preserve command and then finding the cause of your problem, you can use the **recover** command again.

# Chapter 3

# EX EDITOR

# Chapter 3

# EX EDITOR

## INTRODUCTION

This chapter describes the ex editor used on the 3B2 Computer. Ex provides the advanced user a wide range of commands and options, but can also be used by new or casual users who only need a simple editor.

When using the ex editor, all commands must be entered on a command line. The command line is identified by a colon ":" on a line by itself. Commands entered on the command line can affect the line you are on in the file (current line), a specified set of lines, or the entire file.

Most ex editor command names are English words, that can be abbreviated. When an abbreviation conflict is possible, the more commonly used command has the shorter abbreviation. For example, since **substitute** is abbreviated by **s**, **set** is abbreviated by **se**.

The ex editor does not directly change the file being edited. Instead, it works on a copy of the file stored in a temporary memory location called the buffer. The edited file is not changed until you write the changes from the buffer to the edited file.

This editor description assumes that you know how to log on to the computer. If you do not, refer to the *AT&T 3B2 Computer Owner/Operator Manual.*

For additional information on the ex editor, see the manual pages in the *AT&T 3B2 Computer User Reference Manual.*

## CURRENT LINE DEFINITION

The term "current line" is referred to throughout this chapter. The current line is the line in the file you are now on. Each time you move to a different line in the file, that line becomes the current line. Whenever a command is given, the current line is used as a reference point. Any command that is not directed at any specific line is executed against the current line. You should always know what line is the current line, or you could mess up the file.

# GETTING STARTED

The ex editor can be used to create a new file or to change an existing file. To execute ex, you must be logged onto the computer. After the **$** or # prompt is displayed, you can begin working with the ex editor.

## Creating a New File

To create a new file, you will need to type **ex** followed by a space and then the name of the file you wish to create. Execute the command by depressing the carriage return <*CR*>. For example:

```
$ ex filename<CR>
" filename" [New file]
:
```

If you did not enter the command correctly, you will receive a usage message indicating an incorrect command syntax was used. You will need to re-enter the command correctly.

If you entered the **ex** command without a filename, the editor will still create a new file. However, when you decide to write the file into memory you will be prompted for a filename. See "Writing the Buffer Into the File."

When the **ex** command is executed, a colon ":" is displayed. The colon identifies the command line and shows that the ex editor is ready to accept your input commands.

## Entering Text

Most ex commands have two forms: a word that describes what the command does and an abbreviation of the word. You can use either form. Many beginners find the full command name easier to remember, but after some practice use the abbreviation. The command to input text is **append**, that may be abbreviated **a**. Enter **append** after the colon on the command line and then depress the carriage return.

**:append**<*CR*> or **:a**<*CR*>

The ex editor is now in the *text input mode* (append mode). The colon is no longer displayed on the command line, and this is your signal that you may begin entering lines of text. Anything that you type on your terminal, except a period on a line by itself, is entered into the buffer. If the error message:

```
Not an editor command
```

is displayed, check to see what you entered incorrectly and then enter the command again.

> ***Note:*** The computer considers a blank space to be a character. Be careful not to input blanks into lines of text unless you mean for them to be there.

## Leaving the Input Mode

To leave the input mode, simply enter a period "." on a line by itself and depress the carriage return. This is the signal that you want to stop inputting text. After receiving a period on a line by itself, ex will re-enter the command mode and display the command line prompt ":".

The text just entered is now stored only in the buffer. If you wish, you can make changes to the text. Making changes is discussed throughout the remainder of this chapter.

## Writing the Buffer into the File

The buffer is only temporary storage for the file. Now that you have entered text in the buffer, you need to write the buffer to the file. This is the only way to save new text from one editing session to another. To write the contents of the buffer to the file, use the **write** command (abbreviated **w**).

:write<*CR*> or :w<*CR*>

Ex will then copy the buffer into the file. If the file does not yet exist, a new file will be created and a message will be given indicating that it is a new file. The newly created file will be given the name specified when you entered the editor, "filename". To confirm that the file has been successfully written, the editor will repeat the filename, and give the number of lines and the total number of characters in the file. The buffer remains unchanged, so you can make further changes if you want to.

Ex must have a filename to use before it can write a file. Therefore, if you did not show the name of the file when you began the editing session, ex will issue the message:

No current filename

when you give the **write** command. If this happens, simply re-enter the **write** command and specify the filename. Here you would enter:

:write **filename**<*CR*> or :w **filename**<*CR*>

This will write the buffer to a file named "filename".

## Quitting the Editor

When you have finished editing the file and you are ready to return to the UNIX System, enter the **quit** command (abbreviated **q**).

:quit<*CR*> or :q<*CR*>

This returns you to the UNIX System unless you forget to write the buffer

to the file. When this happens, you will receive a message reminding you to write the file. A quick way to **write** and **quit** the ex editor is with the single command:

:wq<*CR*>

If for some reason you do not want to save the changes, enter the command:

:q!<*CR*>

This will quit ex and leave the file unchanged from the last write command.

## Editing an Existing File

To edit the contents of an existing file named "file1", you begin by issuing the command:

```
$ ex file1<CR>
"file1" 150 lines, 4285 characters
:
```

This places a copy of the file in a buffer, and displays how many lines and characters are in the file. A colon ":" will then be displayed, this is the command line.

> ***Note:*** If you do not give a filename, ex will create a new file instead of editing the file you want.

After the file description and the colon ":" are displayed, enter a **1** on the command line followed by a carriage return. This will make the first line in the file the current line. The editing process is described throughout the remainder of this chapter.

The procedure for saving changes to the buffer is described in "Writing the Buffer Into the File." The procedure for quitting the editor is described in "Quitting the Editor."

## DISPLAYING LINES IN THE FILE

The procedures for displaying lines of a file when using the ex editor are the same as for the edit editor. Refer to the procedures given in Chapter 2.

## MOVING AROUND IN THE FILE

The procedures for moving around in a file when using the ex editor are the same as for the edit editor. Refer to the procedures given in Chapter 2.

## MAKING CORRECTIONS TO THE FILE

The procedures for making corrections to a file when using the ex editor are the same as for the edit editor. Refer to the procedures given in Chapter 2.

# FILE MANIPULATION

## Writing the Buffer to Another File

The **write** command (abbreviated **w**) allows you to write all or part of the buffer to a new file. This allows you to keep copies of the buffer in various states of change. To write the whole buffer to another file, use the write command and the name of the file. For example:

**:write filename**<*CR*> or **:w filename**<*CR*>

Be careful when naming the file. If you use an existing filename, the editor will display the message:

```
"filename" File exists - use "w! filename" to overwrite
```

When this occurs, you can either use a different filename, or use the **w!** command to overwrite the file. If you overwrite the file, the information being overwritten is no longer accessible.

If you only want to write part of the buffer to another file, you must specify the beginning and ending lines you want to write. For example:

**:85,$w save**<*CR*>

will write lines 85 through the end of the buffer to the file named *save*. The write command does not change the buffer.

## Reading Another File Into the Buffer

The **read** command (abbreviated **r**) allows you to input the contents of another file into the buffer without destroying the text already there. To use the read command, first move to the line where you want the file appended. Then enter the read command using the following format:

**:read filename**<*CR*> or **:r filename**<*CR*>

Ex will append a copy of the file after the current line, and issue a message

stating the name of the file, the number of lines, and the number of characters that were inserted.

## Obtaining Information About the Buffer

Ex maintains a record of the current information about the buffer. To access this information, enter the **file** command (abbreviated **f**). Ex displays the filename, your current position, and the number of lines in the buffer. If the contents of the buffer have been changed since the last time the file was written, the editor will tell you that the file has been modified. It also displays what per cent of the way you are through the buffer. For example, enter the command:

:f<*CR*>

The computer will respond with a message such as:

```
"filename" [Modified] line 15 of 75 --20%--
```

***Note:*** After you save the changes by writing the buffer to the file, the buffer will no longer be considered modified.

## Read-Only Mode

If you want to look at a file you have no intention of changing, you can execute ex in the read-only mode. This mode protects you from accidentally overwriting the file. The read-only option can be set by using the **-R** command line option, by the **view** command line invocation, or by setting the read-only option. It can be cleared by setting the **noreadonly** mode. (See "OPTION DESCRIPTION.") It is possible to write, even while in the read-only mode, by writing to a different file or by using the **:w!** command.

## Editing More Than One File

The ex editor is normally used to edit the contents of a single file, whose name is recorded in the current file. However, if you want to access another file without quitting ex, you can use the **e** command. For example:

:e **file2**<*CR*>

where "file2" is the name of the second file. This allows you easy access to both files. The current file is always the one currently being edited. The alternate file is the other file you have access to.

When you want to change to the alternate file, use the **e** command with the filename. Each time you use the **e** command to change files, the file you name becomes the current file and the file you leave becomes the alternate file.

When using the **e** command within the editor, normal shell expansion conventions such as "f*1" for "file1" may be used. In addition, the character **%** can be used in place of the current filename, and the character # in place of the alternate filename. For example:

:e #<*CR*>

will cause the alternate file to become the current file, and the current file will become the alternate file. This makes it easy to deal alternately with two files and eliminates the need for retyping the filename.

## Editing Multiple Files and Using Named Buffers

When you have several files that you want to edit without actually leaving and re-entering the ex editor, you can list these files in your ex command. After receiving the command line prompt ":", you can edit file1 as described in this chapter. The remaining arguments are placed with the first file in the argument list. To display the current argument list, enter the **args** command on the command line. To edit the next file in the argument list, enter the **next** command on the command line. The following example shows how to enter three files with the ex command,

how to display the argument list, and how to change to the next file to be edited:

```
$ ex file1 file2 file3<CR>
3 files to edit
" file1"  xxx lines, xxxx characters
:args<CR>
[file1] file2 file3
:next<CR>
" file2"  xxx lines, xxxx characters
:
```

The argument list can be changed by specifying a list of filenames with 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 amount 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.

## ISSUING UNIX SYSTEM COMMANDS

The procedure for issuing UNIX System commands from the ex editor is exactly the same as for the edit editor. Refer to the procedure given in Chapter 2.

# RECOVERING LOST TEXT

## Undoing the Last Command

The **undo** command (abbreviated **u**) is able to reverse the effects of the last command executed that changed the buffer. This enables you to restore the buffer after making an editing mistake. To execute the undo command enter:

:**undo**<*CR*> or :**u**<*CR*>

The undo command only works on commands such as; append, insert, delete, change, move, copy, and substitute. You can also undo an undo if you decide to keep the change. Commands that do not affect the buffer such as: write, edit, and print cannot be undone.

## Recovering Lost Files

If the system crashes, you can recover the contents of the buffer by using the **recover** command. **The recover command cannot be abbreviated.** You will normally receive mail the next time you log in giving you the name of the file that was saved for you. You should then change to the directory containing the file being edited when the system crashed. Then, access the file by entering:

:**ex filename**<*CR*>

replacing "filename" with the name of the lost file. Once in the editor, enter:

:**recover filename**<*CR*>

Recover is sometimes unable to save the entire contents of the buffer, so always check the contents of the saved buffer before writing it back to the original file.

If something goes wrong with the editor when you are using it, do not leave the editor. You may be able to save your work by using the **preserve**

command (abbreviated **pre**). This saves the buffer as if the system had crashed.

If you are writing the buffer into the file and you get the message:

```
Quota exceeded
```

you have tried to use more disk space than you are allotted. When this happens, it is likely that only part of the buffer was written into the file. When this happens, you should escape to the UNIX System using the **sh** command and remove some files you do not need. Then, try writing the file again. If this is not possible, enter the command:

**:preserve**<*CR*>

and then get help from the person who is administrating the system. **Do not quit the editor or your buffer will be lost**.

After using the preserve command and then finding the cause of your problem, you can use the **recover** command again.

## 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. To recover a file you can use the **-r** option. For example: If you were editing the file "filename", you should change to the directory where you were when the crash occurred, and give the command:

**:ex -r filename**<*CR*>

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

You will normally get mail from the system telling you when a file has been saved. The command **ex -r** will print a list of the files that have been saved for you.

### Errors and Interrupts

When errors occur, ex rings the terminal bell (or flashes the terminal screen) and prints an error message. If the primary input is from a file, editor processing will end. If an interrupt signal is received, ex will display the message:

```
Interrupt
```

and returns to its command level. If the primary input is a file, ex will exit.

## COMMENT LINES

It is possible to give editor commands that 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 end of commands except in cases where they could be confused as part of the text (shell escapes, substitute commands, 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 ended by a |.

# OPTION DESCRIPTION

The options that you can set when using the ex editor are the same as for the vi editor. For a listing and a description of these options, see Chapter 4.

## Ex Command Line Options

Instead of just entering the standard ex editor, you can use many options that are sometimes helpful. An example of a command line showing the proper format for using options is shown below.

**ex** [-][**-v**][**-t** *tag*][**-r**][**-w***n*][**-R**][+*command*] *filename*

These options are given in the following list, along with a short description of their function.

- The - command line option suppresses all interactive-user feedback and it is useful in processing editor scripts in command files.

**-v** The **-v** option is equivalent to using vi rather than ex.

**-t** The **-t** option is equivalent to an initial *tag* command, editing files containing tag and positioning the editor at its definition.

**-r** 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.

**-w** The **-w** option sets the default window size to n and is useful on dial-ups to start in small windows.

**-R** The **-R** option sets the read-only option at the start.

+*command*
An argument of the form +*command* tells the editor to begin by executing the specified command. If +*command* is omitted, ex will make the last line of the first file the current line.

*filename*

The *filename* arguments show the file to be edited. More than one filename can be given if several files are to be edited. See "FILE MANIPULATION" for further information on editing multiple files.

# Chapter 4

# VISUAL EDITOR (vi)

PAGE

# Chapter 4

# VISUAL EDITOR (vi)

## INTRODUCTION

This chapter describes the visual editor (vi)* used on the 3B2 Computer. Vi is an interactive text editor that uses the screen of your terminal as a window into the file you are editing. Any changes you make to the file are reflected on the screen.

The vi editor does not directly change the file you are editing. Instead, it makes a copy of the file in a buffer and remembers the file's name. You do not affect the contents of the original file unless you write the changes made back into the original file.

Most vi commands move the cursor around in the buffer. A small set of operators such as **d** for delete and **c** for change alter the text in the buffer. Some of these commands and operators are combined to form operations

* The visual editor (vi) was developed by the Electrical Engineering and Computer Science Department of the University of California, Berkeley Campus.

such as "delete a word" or "change a paragraph." the mnemonic assignment of commands to keys makes the editor command set easy to remember and use.

There are normally several different vi editor commands you can use to get the same results. If you are trying to use vi for the first time, pick a few commands and use them until you no longer have to look them up. Then, gradually try using new commands. You will eventually find more efficient ways of doing the same things. The "CHARACTER FUNCTIONS SUMMARY" at the end of this chapter provides a complete list of vi commands.

This editor description assumes that you know how to log on to the computer. If you do not, refer to the *AT&T 3B2 Computer Owner/Operator Manual.*

For additional information on the vi editor, see the manual pages in the *AT&T 3B2 Computer User Reference Manual.*

## Relations Between vi and ex Editors

The vi editor is actually one mode of editing within the ex editor. When you are running vi, you can escape to the line-oriented editor (ex) by giving the **Q** command. Most ex commands can be invoked separately from vi by first entering a **:** and then the ex command. To execute the command, depress the carriage return.

In rare instances, an internal error may occur in vi. Here, 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 entering the command:

**:x**<*CR*>

If you would want to re-enter vi, you can enter the command:

**:vi**<*CR*>

Experienced users often mix their use of ex command mode and vi command mode to speed the work they are doing. The ex editor is described in Chapter 3.

# GETTING STARTED

## Defining Your Terminal

To use the vi editor, your 3B2 Computer needs to know what type of terminal you are using. The file **/etc/terminfo** contains the parameters of various terminals. Each type of terminal has a unique code assigned to it. To access the information in /etc/terminfo, you need to set the variable "TERM" to the code for your terminal and then export the variable. For example, to tell the computer you are using a TELETYPE* Model 5620 terminal, you would need to enter the following commands:

```
$ TERM=5620<CR>
$ export TERM<CR>
$
```

## Setting Up Your Terminal Configuration

Vi will work on many types of video display terminals, and new terminal types can be added to a terminal description file. Before vi can be used on some terminals, the terminal setup parameters will need to be changed. The changes will vary depending on the terminal. For example, the TELETYPE Model 5410 terminal has a settable parameter called "RCVD'LF" that should be set to "INDEX". For instructions on how to change settable parameters, see the manual supplied with the terminal.

> ***Note:*** For more information on setting up your terminal, see the *AT&T 3B2 Computer Owner/Operator Manual.*

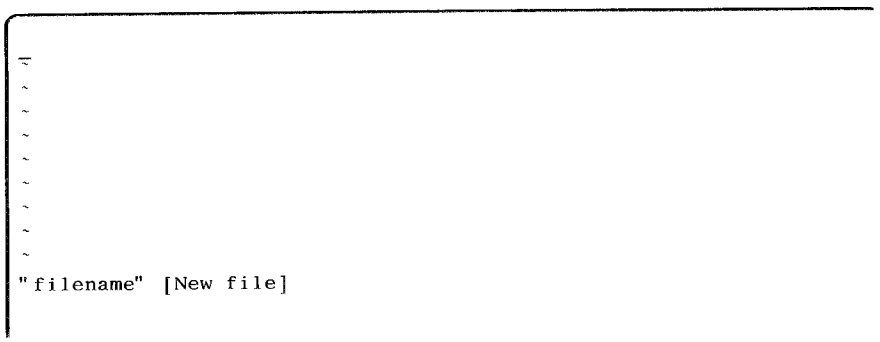
* Trademark of AT&T

## Creating a New File

To create a new file, you will need to type **vi** followed by a space and then the name of the file you wish to create. Execute the command by depressing the carriage return <*CR*>. For example:

**$ vi filename**<*CR*>

will create a file named "filename", clear the screen, and place the cursor at the top of the screen.




Once the vi command is executed, proceed to "Entering Text." If you did not enter the command correctly, you will receive a usage message indicating an incorrect command syntax was used. Reenter the command correctly.

Another problem that can occur is if you gave the system an incorrect terminal code (see GETTING STARTED). The editor may mess up your screen because vi sends control codes for one type of terminal to some other type of terminal. Here, enter the command:

**:q**<*CR*>

This should get you back to the UNIX System shell. Make sure you entered the correct terminal type and then try again.

## Entering Text

To begin inputting text in a file, you must enter the *text insertion mode*. To do this you will need to enter either an **a**, an **i**, or an **o** (not followed by a carriage return). Since these are vi commands, they will not be displayed on the screen. After entering the *text insertion mode*, any characters you type are entered into the buffer.

> ***Note:*** The computer considers a blank space to be a character. Be careful not to input blanks into lines of text unless you mean for them to be there.

## Leaving the Text Insertion Mode

To leave the *text insertion mode*, simply depress the *<ESC>* key. The computer response should be to backspace one character. This will return you to the command mode.

Returning to the command mode does not destroy the text in the buffer. You must return to the command mode to change any other type of editor command.

## Writing the Buffer into the File

The buffer is only temporary storage for the file you are editing. Once you have entered text in the buffer, you need to write the buffer to the file. This is the only way to save new text from one editing session to another. To write the contents of the buffer to the file, use the **write** command (abbreviated **w**).

`:write`*<CR>* or `:w`*<CR>*

Vi will then copy the buffer into the file. If the file does not yet exist, a new file will be created, and a message will be given indicating that it is a new file. The newly created file will be given the name specified when you entered the editor, "filename". To confirm that the file has been successfully written, the editor will repeat the filename, and give the

number of lines and the total number of characters in the file. The buffer remains unchanged, so you can make further changes if you want to.

> ***Note:*** The **:w** command should be used every few minutes if you are happy with the changes you have made. This will keep you from losing all of your changes if you mess up the file or decide you do not like the changes you have made since the last time you wrote the file.

Vi must have a filename to use before it can write a file. If you did not show the name of the file when you began the editing session, vi will not write the file when you give the **write** command. If this happens, simply reissue the **write** command and specify the filename. Here you would enter:

**:write filename**<*CR*> or **:w filename**<*CR*>

This will write the buffer to a file named "filename".

## Quitting the Editor

When you have finished working in the file and you are ready to return to the UNIX System, there are several methods you can use. If you have already written the buffer to the file, enter the command:

**:q**<*CR*>

To write the contents of the buffer back into the file you are editing and then quit the editor, enter the command:

**:wq**<*CR*>, **:x**<*CR*>, or **ZZ** (without depressing <*CR*>)

If for some reason you do not want to save the changes, enter the command:

**:q!**<*CR*>

This will quit vi and leave the file unchanged from the last write command.

## Editing an Existing File

To edit the contents of an existing file named "filename", you begin by issuing the command:

**$ vi filename**<*CR*>

This places a copy of the file in a buffer. The screen should clear and the text of your file should appear on the screen. For example:

```
This is the first line in the file.
This is the second line in the file.
When a tilde (~) is displayed on a line by
itself, it normally means "end-of-the-file".
~
~
~
~
~
~
"filename" 4 lines, 161 characters
```

If the editor printed a "New file" message, you either gave the wrong filename or you are in the wrong directory. Here, you should enter **:q**<*CR*> to get you out of the editor. Check what directory you are in and try entering the command again.

If the editor makes a mess out of your screen, perhaps you gave the system an incorrect terminal type. Here, enter **:q**<*CR*> to get you back to the command level interpreter. If the editor does not respond, try sending an interrupt to it by depressing the <*DEL*>, <*BREAK*>, or <*RUBOUT*> key on your terminal. Then try entering **:q**<*CR*> again. Figure out what you did wrong and try again.

Once you have executed the vi command and you are in the buffer, you may begin moving the cursor around and change the file. Procedures for saving the changes to the buffer are described in "Writing the Buffer into the File." Procedures for quitting the editor are described in "Quitting the Editor."

## Reading an Existing File

If you only want to use the editor to look at a file rather than to make changes, use the command:

```
$ view filename<CR>
```

This will set the read-only option that will prevent you from accidentally overwriting the file. Commands that move the cursor or change the file will execute. However, if you try to use the write command, you will receive the message:

```
"filename" File is read only
```

If you decide that you do want to change the file, you can still write the buffer to the file by entering the command:

```
:w!<CR>
```

# MOVING AROUND IN THE FILE

The vi editor has many commands for moving around in a file. These commands allow you to: scroll through the file; search for a string of characters; or move from page to page, line to line, or character to character. Most of these commands can be preceded by a number to make movement in the file easier. A simple example would be to depress the 5 key and then the return key, this will move the cursor down 5 lines in the file.

> ***Note:*** Searching for a string of characters will not work when preceded by a number.

While reading through this chapter, you will notice that commands such as *<CTRL D>*, *<CTRL L>*, or *<CTRL H>* are used. This refers to commands where it is necessary to depress the control key and one other key at the same time. These are referred to as control characters. This may cause some confusion at first, but should not be a problem when you actually start using the vi editor.

> ***Note:*** When using the vi editor, be careful not to leave the *caps lock* key locked down. Capital letter commands are different from lowercase letter commands and you could accidentally mess up your file. If you do execute the wrong command, you can either use the undo command or quit without writing. (See "RECOVERING LOST TEXT.")

## Scrolling and Paging Through the Screen

Scrolling and paging are two of the ways to move through a file. The main difference is that it is easier to read through a file while scrolling because the screen rolls up or down one line at a time. Paging causes the screen to be blanked each time a new page is displayed.

### *Scrolling*

Scrolling allows you to continuously read through the file you are editing. *<CTRL D>* allows you to scroll down through the file until you release the keys. You can also scroll up through the file by using the *<CTRL U>* command. Some terminals cannot scroll up at all. Depressing *<CTRL 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, you can depress *<CTRL E>* to expose one more line at the bottom of the screen, leaving the cursor where it is. The *<CTRL Y>* command is similar to the *<CTRL E>* command, except that it exposes one more line at the top of the screen.

### *Paging*

Paging is a way to move forward or backward through a file a page at a time. The *<CTRL F>* command will move forward a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file. The *<CTRL B>* command is similar to the *<CTRL F>* command, except that it will move backward a page.

## Cursor Movements

### *Moving Within a Line*

Some commands move the cursor one position at a time, and others move the cursor a word at a time. Preceding numbers may be used with all these commands. Keys that move the cursor a word at a time will wrap around the end of the line to the next line.

These commands are described in the following list:

| | |
|---|---|
| **b** | Moves the cursor to the beginning of the previous word |
| **e** | Moves the cursor to the end of the next word |
| **h** | Moves the cursor one position to the left |
| **l** | Moves the cursor one position to the right |
| **w** | Moves the cursor to the beginning of the next word |
| **B** | Moves the cursor to the beginning of the previous word without stopping at punctuation marks |
| **W** | Moves the cursor to the beginning of the next word without stopping at punctuation marks |
| *<CTRL H>* | Control character that moves the cursor one position to the left |
| backspace | Moves the cursor one position to the left |
| spacebar | Moves the cursor one position to the right. |

***Note:*** On some terminals, the arrow keys will also move the cursor around on the screen. Most experienced users of vi normally prefer the **h**, **j**, **k**, and **l** keys because they are usually right beneath their fingers.

### *Moving To Different Lines*

There are several commands you can use to move the cursor to a different line on the screen. All these commands except **H**, **L**, and **M** take preceding numbers and act on them. These commands are described in the following list:

| | |
|---|---|
| **j** | Moves the cursor down |
| **k** | Moves the cursor up |
| RETURN | Moves the cursor to the first position on the next line |
| + | Moves the cursor to the first nonwhite position on the next line |
| – | Moves the cursor to the first nonwhite position on the previous line |
| **H** | Moves the cursor to the top line of the screen |
| **M** | Moves the cursor to the middle of the screen |
| **L** | Moves the cursor to the last line of the screen |
| *<CTRL N>* | Control character that moves the cursor down a line in the same column |
| *<CTRL P>* | Control character that moves the cursor up a line in the same column. |

### *Moving Through a File*

When working with a file containing text, it is often easier to work in terms of sentences, paragraphs, and sections. The following list describes some useful commands for working with text. Preceding numbers may be used with sentence and paragraph commands.

( Moves the cursor to the beginning of the previous sentence.

) Moves the cursor to the beginning of the next sentence.

> ***Note:*** A sentence is defined to end at a ., **!**, or **?**, and is followed by the end of the line or two spaces. Any number of ), }, ", and ' closing characters may appear after the ., **!**, or **?**, and before the spaces or end of line.

{ Moves the cursor to the beginning of the previous paragraph.

} Moves the cursor to the beginning of the next paragraph.

> ***Note:*** A paragraph begins after each empty line and also at each paragraph macro specified in the *paragraphs* option. The .bp request is also considered to start a paragraph.

[[ Moves the cursor to the beginning of the previous section.

]] Moves the cursor to the beginning of the next section.

> ***Note:*** Sections begin after each macro in the *section* option and each line with a form feed *<CTRL L>* in the first column. Section boundaries are always line and paragraph boundaries.

## Searching Through the File

Another way to position yourself in the file is by having the editor search for a specific string of characters on one line. Type the character / followed by a string of characters for which you want to search. To execute the search, depress the carriage return. For example:

**/character string**<*CR*>

The editor will search from the current position toward the last line in the buffer for the first occurrence of "character string" on one line. The editor will also search backward if you use the **?** character instead of the / character.

If the character string you search for is not present in the file, the editor will display the message:

```
Pattern not found
```

on the last line of the screen and the cursor will return to its initial position.

A search will normally wrap around the end of the file and continue searching until the string is found or the position where the search started is reached. The wrap-around scan feature can be disabled by entering the command:

**:set nowrapscan**<*CR*> or **:set nows**<*CR*>

You can have the editor ignore whether letters are uppercase or lowercase in searches by entering the command:

**:set ignorecase**<*CR*> or **:set ic**<*CR*>

The command **:set noic**<*CR*> turns this option off.

## Repeating Searches

If the first pattern found by the search command is not the one you were searching for, you can search for the next occurrence of the pattern by entering the command:

**n**

The **n** command works with forward and backward searches.

Another way to repeat a search without re-entering the entire command is to enter the search command character (/) or (**?**) followed by a carriage return. The direction of the search is determined by the search character you enter.

## Special Search Characters

Several characters have special meanings when used in specifying searches. These characters will work with all types of searches. They can be used to: match repetitive strings of characters, turn off special meanings of characters, or denote the placement of characters in the line. These characters and their uses are explained below:

. The period matches any single character except the newline (carriage return) character. For example, if a line in your file contains the words "vi editor", or a pattern with any other character between "vi edit" and "r", you could find the line by entering the command:

**:/vi edit.r/**<*CR*>

* The asterisk matches any repeated characters except the first ., \, [, or ~ in that group. For example, if a line in your file contains the pattern "the xxxx editor", you could search for the line by entering the command:

**:/the x* editor/**<*CR*>

[] Brackets are used to enclose a variable set of characters. For example, if you have a file containing the patterns"file2", "file3", or "file4" you could search for the first occurrence of these patterns by entering the command:

**:/file[2-4]/**<*CR*>

$ The dollar sign is interpreted by the editor to mean "end of the line". It is used to identify patterns that occur at the end of a line. For example, if a line in your file ends in the pattern"last character", you could find the line by entering the command:

**:/last character$/**<*CR*>

^ The circumflex (caret) works like "**$**" except it looks for the pattern at the beginning of the line. For example, if a line in your file begins with the pattern "First character", you could find the line by entering the command:

**:/^First character/**<*CR*>

\ The backslash is used to cancel the meaning of the special characters. It should be placed immediately before the character it is to nullify. For example, if a line in your file contains the pattern "This is a $", you could search for it by entering the command:

**:/This is a \$/**<*CR*>

The character $ will be searched for instead of being interpreted as meaning "end of the line".

To search for the characters ., *, \, [, ], $, or ^, you must precede the characters with a backslash. You can also combine these special characters in one search command. For example, .* can be used to search for any string of characters.

## Go To, Find, and Previous Context Commands

The go to (**G**) command allows you to move the cursor to a specific line in the file by using line numbers. For example:

**32G**

will move the cursor to line 32 in the file. If a line number is not used with the **G** command, the cursor will move to the last line in the file.

The find (**f***x*) command locates the next *x* character to the right of the cursor in the current line. For example, to find the next occurrence of the letter **t** you would enter the command:

**ft**

The **;** command repeats the last find command for 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 works like **f**, but searches backward. The **;** also repeats the **F** command.

The previous context " (two back quotes) command allows you to move back to the previous position in the file after a motion command, such as /, **?**, or **G**. This command is often more convenient than using the **G** command or performing a search because no advance preparation is required.

> ***Note:*** If you are near the last line of the file, and the last line is not at the bottom of the screen, the editor will place a ~ character on each remaining line to show the end of the file.

# MAKING SIMPLE CHANGES

## Inputting Text

The vi editor uses append, insert, and open commands to input text into a file. First, use the movement commands described earlier to move the cursor to the position in the file where you want to input text. Then depress the input command you want to use (see list below). Now any characters you type are entered into the buffer. If you are entering more than one line, depress a carriage return whenever you want to start a new line. You can also use the **autowrap** option discussed in "OPTIONS." To stop inputting text, depress the *<ESC>* key. All the commands for inserting text are described in the following list:

- **a** Appends everything you type after the current position of the cursor
- **A** Appends everything you type to the end of the line
- **i** Inserts everything you type before the current position of the cursor
- **I** Inserts everything you type before the first nonblank on the line (inserts before the first character on the line)
- **o** Opens a new line below the position of the cursor
- **O** Opens a new line above the position of the cursor.

### *Erasing Inserted Text*

While inserting text, you can use the *<CTRL H>* or # character to backspace over (erase) the last character typed. To erase the text you have input on the current line, depress the @, *<CTRL X>*, or *<CTRL U>* characters. The *<CTRL 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.

While inserting text, 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. The characters disappear when you depress <*ESC*>. If you want to get rid of the characters immediately, depress <*ESC*> 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 correct something, depress the <*ESC*> key, move the cursor back to the previous line, and then make whatever corrections you want.

### *Continuous Text Input*

When you are typing in large amounts of text, it is convenient to have lines broken near the right-hand margin automatically. You can cause this to happen by entering the command:

**:set wm=10**<*CR*>

This causes all the lines to be broken at a space at least ten columns from the right-hand edge of the screen. The number 10 can be replaced by any number you wish to use.

### *Joining Lines*

If the editor breaks a line and you wish to put it back together, you can tell it to join the lines with the **J** command. You can give the **J** command a count of the amount of lines to be joined (such as **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. If you do not want white space, you can kill it with the **x** command.

## Removing Text

The vi editor allows you to remove text from a file with several versions of the delete command. The commands listed below let you remove any object that the editor recognizes (characters, words, lines, sentences, and paragraphs). You do not need to use the *<CR>* or *<ESC>* keys with these commands. To delete more than one object at a time, you can use numbers with these commands. For example, **5dd** removes five lines of text.

**dd** Delete the current line.

**dw** Delete the current word.

**db** Delete the preceding word.

**d)** Delete the rest of the current sentence.

**d(** 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.

**d}** Delete the rest of the current paragraph.

**d{** Delete the previous paragraph if you are at the beginning of the current paragraph, or delete the current paragraph up to your current position if you are not at the beginning of the current paragraph.

**D** Delete the rest of the text on the current line and leave the cursor on a blank line.

**x** Delete the current character.

**X** Delete the character before the cursor.

***Note:*** To recover text that was accidentally deleted, see "Recovering Lost Text."

## Changing Text

The vi editor allows you to use several different commands to change text in a file. With the commands listed below you can change any object that the visual editor recognizes (characters, words, lines, sentences, and paragraphs). All these commands, except **r**, are ended by depressing the *<ESC>* key. Numbers can be used with these commands to determine how many of the objects to change. For example, the command **2cw** removes two words and then changes to the input mode so new words can be inserted.

**cc** Change a whole line.

**cw** Change the specified word to the following word.

**c)** Change the rest of the current sentence.

**c(** Change the previous sentence if you are at the beginning of the current sentence, or change the current sentence up to your current position if you are not at the beginning of the current sentence.

**c}** Change the rest of the current paragraph.

**c{** 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.

**C** Change the rest of the current line.

**r** Replace a character.

**R** Replace the following characters.

**s** Replace a character with a string.

**S** Replace the current line with a new line.

When you type a change command, the end of the text to be changed is marked with the **$** character to show that a change is now expected up to the **$** character. You are now placed in the insert mode so that anything you type is entered into the buffer. You end the insert mode by depressing *<ESC>*. To summarize, change commands in the visual editor deletes text objects and then places you in the insert mode.

The simplest change that you can make is to change one character. The **r** and the **s** commands can be used for this. If the character is incorrect and is to be replaced by a single character, correct the character by giving the **r***x* command, where *x* is the correct character. If the character is to be replaced by a string of characters, give the **s** (string) *<ESC>* command that substitutes a string of characters for the incorrect character. The **s** command can be preceded with a count of the amount of characters to be replaced.

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 **c**/*string* command allows you to change characters from the current position to the first occurrence of the search string.

***Note:*** To recover text that was accidentally changed, see "Recovering Lost Text."

# 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 the cursor 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 create 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. The **P** command 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 create 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

" *x***y**object

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 the command:

   **" 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 the command:

   **" ap** or **" aP**

   This puts the line at the new location.

## Copying Objects

The yank and put commands can be used to copy characters, words, lines, sentences, or paragraphs. All the object commands can be preceded by a number, that allows you to copy more than one object. This is especially useful when copying characters. Each of the following objects should be experimented with so you understand what happens during a yank and put.

Characters can be copied by typing the yank command and then typing 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 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 the following objects:

| | |
|---|---|
| **$** | Yanks one line from cursor to end of line. |

<*CR*> 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 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 the following objects.

}  Yanks from cursor to end of paragraph in forward direction.

{  Yanks from cursor to beginning of paragraph in reverse direction.

## MOVING TEXT

The blocks of text that can be moved around in the file are: characters, words, lines, sentences, and paragraphs. To move blocks of text from one location to another, use the following procedure:

1. Delete (or change) the information you need to move with one command. It will be saved in an area and appointed to a register.

2. Move the cursor to the location you wish to insert the text just deleted and put it 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.

3. 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.

4. 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

   **" *x*delete object**
   or
   **" *x*change 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 the command:

   **" 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 the command:

   **" 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 *<ESC>* key pressed, the deleted text can be "put" at another location in the file.

# GLOBAL COMMANDS

## Global Searches

When you need to locate all the occurrences of a specific pattern on a line in your file, the global command (**:g**) and a search command (/ or **?**) can be used. The global search command can be used in any of the following formats:

(1) **:[m],[n]g/text**
(2) **:[m],[n]g/text/p**
(3) **:[m],[n]g/text/nu**

The **[m]** represents the line number where the search will start. The **[n]** represents the line number where the search will stop, or **$** that causes the search to continue to the end of your file. If no numbers are entered, all lines in the file will be searched.

- When (1) is entered, the cursor will move to the last occurrence of "**text**".
- When (2) is entered, all the lines containing "**text**" are displayed on the screen.
- When (3) is entered, all the lines containing "**text**" are displayed on the screen. Line numbers will be displayed with each line.

In global searches, a **?** substituted for the / will have the same affect. The special characters described in "Special Search Characters" can be used in global search commands.

## Global Substitutes

The global substitute command can be used when the same change needs to be made in several places in the file. The command can be executed against a range of lines or against the whole file. The following formats can be used for global substitutes:

(1) **:[m],[n]g/text/s//newtext**
(2) **:[m],[n]g/text/s//newtext/p**
(3) **:[m],[n]g/text/s//newtext/c**

The **[m]** represents the line number where the search will start. The **[n]** represents the line number where the search will stop. **$** can be used to represent the end of your file. If no numbers are entered, all lines in the file will be searched.

- When (1) 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 (2) 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 (3) is entered, you are in a "prompt" mode. The "prompt" mode will allow you to decide if you want to make the substitution. 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 a *<CR>*, "**newtext**" will be substituted for "**text**" in the file. The next line containing "**text**" will then be displayed with ^'*s replacing* "**text**". If you decide not to make the substitution, type a *<CR>* and the next line with "**text**" will be displayed. The line displayed may appear as follows:

The ^^^^ of this sentence needs to be changed.

The special characters described in "Special Search Characters" can be used in the search part of the global substitution command.

## REPEATING ACTIONS WITH THE . COMMAND

Vi provides a timesaving command, called the "dot" command. The "dot" command allows you to repeat the last command that changed the buffer by placing the cursor at the location you wish to repeat the command and entering a:

.

The actions that can be repeated using the . command are append, insert, open, delete, change, and put. An example of how to use the dot command would be to insert a line of text in a file and then depress the <*ESC*> key. Then move the cursor to a different location in the file and enter a . "dot". Vi will repeat the previous insert command and insert the line of text here also.

If you want to place text at another location that is in a named register after doing a put, you can save time by using the . command. However, if you executed a put command that is associated with an "unnamed" register, the . command should not be used. This is because the text in the unnamed register may not be the same.

# FILE MANIPULATION

## Writing the Buffer to Another File

The **write** command (abbreviated **w**) allows you to write all or part of the buffer to a new file. This allows you to keep copies of the buffer in various states of change. To write the whole buffer to another file, simply use the write command and the name of the file. For example:

```
:write filename<CR> or :w filename<CR>
```

Be careful when naming the file. If you use an existing filename, the editor will display the message:

```
"filename" File exists - "w! filename" to overwrite
```

When this occurs, you can either use a different filename, or use the **w!** command to overwrite the file. If you overwrite the file, the information being overwritten is no longer accessible.

If you only want to write part of the buffer to another file, you must specify the beginning and ending lines you want to write. For example,

```
:85,$w save<CR>
```

will write lines 85 through the end of the buffer to the file named *save*.

The write command does not change the buffer. The editor will display the name of the file "save" that you have copied into, the number of lines, and the number of characters entered into the file "save". If no numbers are entered, the entire file you are in will be copied to the filename entered.

Sometimes it is necessary to append information onto the end of a file that already exists. For example, if you wanted to append several lines to the file "save", you could use the command:

**:12,25w** >>**save**<*CR*>

The editor will display the name of the file "save", the number of lines, and the number of characters added to the file.

## Reading Another File into the Buffer

While using **vi**, it may be necessary to copy another file into the file you are editing. This can be done 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. The format for this command is:

**:[n]read filename**<*CR*> or **:[n]r filename**<*CR*>

**[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 named "test" to follow line 10 in your file, enter the command:

**:10r test**<*CR*>

## Reading the Output From UNIX System Commands into the Buffer

There are two commands that you can use to put the output from a UNIX System command into a file. The only difference between the two commands is that one inserts the text between lines and the other replaces the current line with the text.

To insert the output from a UNIX System command between two lines, position the cursor where you want the text and execute the command:

**:r !cmd**<*CR*>

where "cmd" is the UNIX System command. The inserted text will be displayed on the screen. This command will also allow you to use a line number instead of positioning the cursor where you want the text inserted.

If you want to replace a line in the buffer with the output of a UNIX System command, position the cursor on that line and execute the command:

**!!cmd**<*CR*>

where "cmd" is the UNIX System command. Only the current line will be replaced by the inserted text. The inserted text will be displayed on the screen.

## Changing Files in the Editor

The vi editor is normally used to edit the contents of one file, whose name is recorded as the current file. However, you can edit a different file without leaving the editor by using the command:

**:e filename**<*CR*>

where "filename" is replaced by the name of the file to which you want to change. This command allows you easy access to both files, because vi does not have to be executed again.

When you are accessing two files, the file you are editing is always considered the current file, and the other file is considered the alternate file. When you want to change to the alternate file, use the **e** command with the filename. Each time you use the **e** command to change files, the file you name becomes the current file and the file you leave becomes the alternate file.

When using the **e** command within the editor, normal shell expansion conventions, such as "f*1" for "file1", may be used. In addition, the character **%** can be used in place of the current filename and the character # in place of the alternate filename. For example:

:e #<*CR*>

will cause the alternate file to become the current file and the current file will become the alternate file. This makes it easy to deal alternately with two files and eliminates the need for retyping the filename.

If you have not written the current file, the editor will display the message:

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

and delay editing the other file. You can either give the **:w** command to write the file or **:e! filename** if you want to discard the changes to

the current file and begin editing the next file. To have the editor automatically save the changes, you should include **set autowrite** in your EXINIT and use the **:n** command instead of the **:e** command.

If you want to edit the same file (start over), give the **:e!** command. These commands should be used carefully because once the changes are discarded they cannot be recovered.

## Editing Multiple Files and Using Named Buffers

When you have several files that you want to edit without actually leaving and re-entering the vi editor, you can list these files in your vi command. For example, if you enter the command:

**vi file1 file2 file3**<*CR*>

the computer will respond with a message such as:

```
3 files to edit
"file1" xxx lines, xxxx characters
```

The current file "file1" can now be edited. The remaining arguments are placed with the first file in the argument list. To display the current argument list, enter the command:

**:args**<*CR*>

The computer will respond with the message:

```
[file1] file2 file3
```

The next file in the argument list may be edited by entering the command:

**:next**<*CR*> or **:n**<*CR*>

If you have already written the buffer to the file, the computer will respond with a message such as:

```
"file2" xxx lines xxxx characters
```

If you use the **next** command regularly, you may want to set the **autowrite** option.

The argument list can be changed by specifying a list of filenames with the **next** command. These names are expanded with the resulting list of names becoming the new argument list, and vi edits the first file on the list.

For saving blocks of text while editing, and especially when editing more than one file, vi has a group of named buffers. These are similar to the normal buffer, except that only a limited amount 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

If you want to look at a file that you have no intention of changing, you can execute vi in the read-only mode. This mode protects you from accidentally overwriting the file. The read-only option can be set by using the **-R** command line option, by the **view** command line invocation, or by setting the read-only option. It can be cleared by setting the **noreadonly** mode. (See "OPTIONS.") It is possible to write, even while in the read-only mode, by writing to a different file or by using the **:w!** command.

## Obtaining Information about the Buffer

You can determine the state of the file by using the *<CTRL G>* command. The editor will show you the name of the file, the number of the current line, the number of lines in the buffer, and the percentage of the way through the buffer that the cursor is located. A sample response would be:

```
"filename" [Modified] line 1048 of 3096 --33%--
```

***Note:*** After you save the changes by writing the buffer to the file, the buffer is no longer considered modified.

## ISSUING UNIX SYSTEM COMMANDS

Vi allows you to execute UNIX System commands by entering commands of the form

**:!cmd**<*CR*>

where "cmd" represents the command you want to execute. Once the command has executed, the computer will issue the message:

```
[Hit return to continue]
```

You can then depress the carriage return to continue editing or enter the **:!** command to issue another UNIX System command.

If you need to execute more than one UNIX System command enter:

**:sh**<*CR*>

The computer will respond with the shell prompt (**$**). When you have finished executing UNIX System commands, enter a <*CTRL d*>. This will return you to the vi editor.

***Caution: Be sure to write the buffer into the file before escaping to the UNIX System. The editor will normally save the buffer, but it will issue a message to remind you.***

# RECOVERING LOST TEXT

## Undoing the Last Command

The **undo** command (abbreviated **u**) is able to reverse the effects of the last command executed. Undo can often rescue the buffer from a disastrous mistake. To execute the undo command enter:

**:undo**<*CR*> or **:u**<*CR*>

The undo command only works on commands that change the buffer, such as — append, insert, delete, change, move, copy, and substitute. You can also undo an undo command if you decide to keep the change. Commands such as write, edit, and print cannot be undone.

The **U** command works like the **u** command, except that it returns the current sentence to its original state.

## Recovering Lost Lines

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:

" *n***p**

The " tells 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 command 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 .

command will repeat the last change. As a special case, when the last command refers to a numbered text register, the . command increments the number of the register before repeating the put command. Thus, a sequence of the form

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

will, if repeated long enough, show all the deleted text that was saved. Omit the **u** commands and place all 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 instead of after the cursor.

## Recovering Lost Files

If the system crashes, you can recover most of the work you were doing. You will normally receive mail the next time you log in giving you the name of the file that has been saved for you. To recover the file, change to the directory where you were when the system crashed and give a command of the form:

```
$ ex -r filename<CR>
```

replacing "filename" with the name of the file that you were editing. This will recover your work almost at the point where you left off.

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

```
$ ex -r<CR>
```

If there is more than one instance of a particular file saved, the editor gives you the newest instance each time you recover it. Therefore, you can get an older saved copy back by first recovering the newer copies.

For the "recover lost file" command to work, vi must be correctly installed and the mail program must exist to receive mail.

## MARKING LINES

The vi editor allows you to mark lines in the file with single letter tags and return to these marks later by naming the tags. For example, mark the current line with an **a** by entering the command:

**ma**

Then, move the cursor to a different line using any commands you like and enter the command:

**'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 marked line. Here, use the form *'x* rather than *`x*. Used without an operator, *'x* will move to the first nonwhite character of the marked line. The '' moves to the first nonwhite character of the line containing the previous context mark ''.

## WORD ABBREVIATIONS

Word abbreviation allows you to type a short word and have it expanded into a longer word or words. The commands are:

**:abbreviate** (or **:ab**)
and
**:unabbreviate** (or **:una**)

and have the same syntax as **:map**. For example:

**:ab ecs Engineering and Computer Sciences**<*CR*>

causes the word "ecs" to always be changed into the phrase "Engineering and Computer Sciences." Word abbreviation is different from macros in that only whole words are affected. If "ecs" 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.

## 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 to your terminal, use the <*CTRL L*> command to refresh the screen.

If you want to place a certain line on the screen at the top, middle, or bottom of the screen, you can position the cursor to that line and use the **z** command followed by its argument. The following list describes the three possible uses of the **z** command:

**zz** Places the line at the top of the screen

**z.** Places the line at the center of the screen

**z-** Places the line at the bottom of the screen.

# 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.

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 *<CTRL R>* command.

## Line Numbers

Vi allows you to place line numbers before each line on the display. To set the line number option, enter the command:

```
:set nu<CR>
```

To remove the line number option, enter the command:

```
:set nonu<CR>
```

## List All Characters on a Line

You can have tabs represented as ^I and the ends of lines shown with $ by entering the command:

```
:set list<CR>
```

To remove the display of tabs and ends of lines enter the command:

```
:set nolist<CR>
```

Lines consisting of only the ~ character are displayed when the last line of the file is in the middle of the screen. These represent physical lines that are past the logical end of the file.

## MACROS

The **vi** editor allows you to create macros 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, 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<CR>*

  mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke (either one character or one function key). It must be entered within 1 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 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 *<CTRL v>* (it may be necessary to double the *<CTRL v>* if the map command is given inside **vi** rather than in **ex**). Spaces and tabs inside the *rhs* need not be escaped. To make the **q** key write and exit the editor, enter:

  **:map q** :wq*<CTRL v><CTRL v><CR> <CR>*

  this means that whenever you type **q**, it will be as though you

had typed **:wq**<*CR*>. A <*CTRL v*> is needed because without it the <*CR*> would end the : command rather than becoming part of the *map* definition. There are two <*CTRL v*>'s because from within **vi**, two <*CTRL v*>'s must be typed to get on. The first <*CR*> is part of the *rhs*, the second ends the : command.

Macros can be deleted with

**:unmap** *lhs*

If the *lhs* of a macro is **#0** through **#9**, the particular function key is mapped instead of the 2-character # sequence. So that terminals without function keys can access such definitions, the form **#x** will mean function key *x* on all terminals (and need not be typed within 1 second). The character # can be changed by using a macro in the usual way:

**:map** <*CTRL v*><*CTRL v*><*CTRL i*> #

to use tab, for example. This will not affect the *map* command, that still uses #, but affects the invocation from *visual* mode.

The **undo** command will reverse all the changes made by a macro call as a unit.

Placing an **!** after the word map causes the mapping to apply to *text input* mode rather than *command* mode. Thus, to arrange for <*CTRL t*> to be the same as four spaces, type

**:map** <*CTRL t*><*CTRL v*>␣␣␣␣

where ␣ is a blank. The <*CTRL v*> is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

# OPTIONS

## Setting Options

There are three kinds of options: numeric, string, and toggle. Numeric and string options are set by a statement of the form:

**:set** *option=value<CR>*

Toggle options can be set or not set by statements of the forms:

**:set** *option<CR>*
and
**:set no***option<CR>*

These options can be placed in your **EXINIT** in your environment or given while you are running vi by preceding them with a : and following them with a *<CR>*.

You can get a list of all options that you have changed with the command:

**:set***<CR>*

or the value of a single option with the command:

**:set** *option* **?***<CR>*

A list of all possible options and their values is generated by the command:

**:set all***<CR>*

Set can be abbreviated **se**. Multiple options can be placed on one line, for example:

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

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 done 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. 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
```

this establishes the **set** command options *autoindent*, *autowrite*, *terse*, makes @ delete a line (the first **map**), and makes # delete a character (the second **map**). One way to have the commands execute every time you enter the vi editor is to put the line in the file *.exrc* in your home directory. Another way to execute the commands automatically is to place the string 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
```

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

## List of Options

The editor has a set of options that can be useful. Some of these options have been mentioned earlier. They are as follows:

**autoindent, ai** (default: noautoindent)
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, the editor 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.

Autoindent 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 aligning with the first nonwhite character of the previous line. To back the cursor to the preceding tabstop, type *<CTRL 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 *<CTRL 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 a ^ and immediately followed by a *<CTRL 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 *<CTRL 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: autoprint)
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: noautowrite)
Causes the contents of the buffer to be written to the current file if you have modified it and enter a **next**, **rewind**, **tab**, or **!** command, or a *<CTRL ↑>* (switch files) or *<CTRL ]>* (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 is 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 similar to the ~ command instead of like the **&** command.

**errorbells,eb** (default: noerrorbells)
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.

**flash, fl** (default: flash)
Errors or illegal inputs respond by flashing the screen instead of ringing the bell in the terminal. On terminals that do not have flash capability, the bell will still ring.

**hardtabs, ht** (default: hardtabs=8)
Gives the boundaries on what terminal hardware tabs are set (or on what the system expands tabs).

**ignorecase,ic** (default: noignorecase)
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.

**list** (default: nolist)
All printed lines will be displayed showing hidden characters such as tabs and end-of-lines.

**magic** (default: magic)
If *nomagic* is set, the amount 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.

**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 LIppIpipbp)
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 (:).

**readonly** (default: noreadonly)
Sets the editor so you cannot accidentally change the file.

**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 high speeds.

**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 changes more than the specified amount of lines will provide feedback on 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=½ window)
Determines the amount of logical lines scrolled when an end-of-file is received from a terminal input in *command* mode, and determines the amount of lines printed by a *command* mode **z** command (double the value of *scroll*).

**sections** (default: sections=NHSHH HUnhsh)
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: shell=/bin/sh)
Gives the path name of the shell forked for the shell escape command **!** and by the **shell** command. The default is taken from SHELL in the environment, if present.

**shiftwidth, sw** (default: shiftwidth=8)
Gives the width a software tabstop used in reverse tabbing with *<CTRL d>* when using *autoindent* to append text and by the shift commands.

**showmatch, sm** (default: noshowmatch)
In *open* and *visual* modes when a ) or } is typed, it moves the cursor to the matching ( or { for one second if this matching character is on the screen.

**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: tabstop=8)
The editor expands tabs in the input file to be on *tabstop* boundaries for the purposes of display.

**taglength, tl** (default: taglength=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.

**timeout** (default: notimeout)
Set a time limit for the execution of an editor command.

**ttytype=**
Terminal type defined to system for visual mode. Can be defined before entering visual editor by TERM=type.

**warn** (default: warn)
Warn if there has been "[No write since last change]" before a **!** command escape.

**window** (default: window=speed dependent)
The amount 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: wrapscan)
Searches that use regular expressions in addressing will wrap around past the end of the file.

**wrapmargin, wm** (default: wrapmargin=0)
Defines a margin for automatic wrapover of text during input in *open* and *visual* modes.

**writeany, wa** (default: nowriteany)
Inhibit checks normally made before **write** commands, allowing a write to any file that the system protection mechanism will allow.

## 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. Usually, uppercase and lowercase *<CTRL>* characters do the same action.

*<CTRL @>* Not a command character. If typed as the first character of an insertion, it is replaced with the last text inserted; and the insert ends. Only 128 characters are saved from the last insert; if more characters have been inserted, the mechanism is not available. A *<CTRL @>* cannot be part of the file owing to the editor implementation.

*<CTRL a>* Unused.

*<CTRL b>* Backward window. A count specifies repetition. Two lines of continuity are kept, if possible.

*<CTRL c>* Unused.

*<CTRL d>* As a command, it scrolls down a half window of text. A count gives the amount of (logical) lines to scroll and the count is remembered for future *<CTRL d>* and *<CTRL u>* commands. During an insert, it backtabs over *autoindent* white space at the beginning of a line. This white space cannot be backspaced over.

*<CTRL e>* Exposes one more line below the current screen in the file, leaving the cursor where it is, if possible.

*<CTRL f>* Forward window. A count specifies repetition. Two lines of continuity are kept, if possible.

*<CTRL g>* Equivalent to **:f** *<CR>*, printing the current filename, 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.

*<CTRL h>* (BS)
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.

*<CTRL i>* (TAB)
Not a command character. When inserted, it prints as some amount 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 tabstops is controlled by the *tabstop* option.

*<CTRL j>* (LF)
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 *<CTRL n>*.

*<CTRL k>* Unused.

*<CTRL l>* The ASCII form feed 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.

*<CTRL m>* (*<CR>*)
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 *<CR>* causes the insert to continue onto another line.

*<CTRL 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 *<CTRL j>*.

*<CTRL o>* Unused.

*<CTRL p>* Same as **Up arrow**. It moves the cursor one line up. A synonym is **k**.

*<CTRL q>* Not a command character. In *text input* mode, *<CTRL q>* quotes the next character, the same as *<CTRL v>*, except that some TELETYPE drivers will delete the *<CTRL q>* so that the editor never sees it.

*<CTRL r>* Redraws the current screen eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line.

*<CTRL s>* Unused. Some TELETYPE drivers use *<CTRL s>* to suspend output until *<CTRL q>* is invoked.

*<CTRL t>* Not a command character. During an insert with *autoindent* set and at the beginning of the line, it inserts *shiftwidth* white space.

*<CTRL u>* Scrolls the screen up (inverse of *<CTRL d>*). A count gives the amount of (logical) lines to scroll, and the count is remembered for future *<CTRL d>* and *<CTRL u>* commands. The previous scroll amount is common to both. On a dumb terminal, *<CTRL u>* will often require clearing and redrawing the screen further back in the file.

*<CTRL 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.

*<CTRL 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 *<CTRL h>*).

*<CTRL x>* Unused.

*<CTRL 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 *<CTRL u>*.

*<CTRL z>* Unused.

*<CTRL [>* (*<ESC>*)
Cancels a partially formed command (such as a **z** when no following character has yet been given), ends inputs on the last line (read by commands such as **:**, **/**, and **?**), and ends insertions of new text into the buffer. If an *<ESC>* is given when in the command state, the editor rings the bell or flashes the screen. Therefore, you can press *<ESC>* 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, type *<ESC a>* and then the material to be input; the material will be inserted correctly whether or not you were in *insert* mode when you started.

*<CTRL e>* Unused.

*<CTRL ]>* Searches for the word that is after the cursor as a tag. It is equivalent to typing **:ta**, this word, and then a *<CR>*.

*<CTRL ↑>* Equivalent to **:e #***<CR>*, returning to the previous position in the last edited file, or editing a file that you specified if you got a "No write since last change" diagnostic and do not want to have to type the file name again. You have to do a **:w** before *<CTRL ↑>* will work in this case. If you do

not wish to write the file, enter **:e!** #<*CR*> instead.

<*CTRL _*> Unused. Reserved as the command character for the TEKTRONIX* 4025 and 4027 terminals.

SPACE 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 ended by <*CR*>. Doubling **!** and preceding it by a count causes count lines to be filtered; otherwise, the count is passed on to the object after the **!**. Thus **2!}***fmt*<*CR*> reformats the next two paragraphs by running them through the program *fmt*. 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** through **9**) used for saving deleted text and named buffers (**a** through **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.

**$** Moves to the end of the current line. If the **:se list**<*CR*> command is used, then the end of each line will be shown by printing a **$** 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.

---

* Registered Trademark of Tektronix, Inc.

**%** Moves to the parenthesis (**(**) or brace ({) that precedes or follows the parenthesis or brace at the current cursor position.

**&** A synonym for **:&**<*CR*>, 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** through **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. A sentence ends at a ., **!**, or **?** followed by either the end of a line or by two spaces. Any amount 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 may be used before **(** to advance more than one sentence.

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

***** Unused.

**+** Same as <*CR*> when used as a command.

**,** Reverse of the last **f**, **F**, **t**, or **T** command, looking the other way in the current line. Especially useful after typing too many **;** characters. A count repeats the search.

**-** Retreats to the previous line at the first nonwhite character. This is the inverse of + and <*CR*>. 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 <*CR*>, and the cursor moves to the beginning of the last line to show that the search is in progress. The search may be ended with a <*DEL*> or <*RUB*>, or by backspacing when at the beginning of the bottom line returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.

When used with an operator, the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern, you can force whole lines to be affected. To do this, give a pattern with a closing / 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 \ to get them to work as you would expect.

**0** Moves to the first character on the current line. Also used, when forming numbers.

**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 ends with a <*CR*>, and the command is then executed. If you accidentally type **:**, you can return to where you were by typing <*DEL*> or <*RUB*>.

**;** 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.

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

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

**@** A macro character. Since this is the kill character, you must escape it with a \ to type it in during *text input* mode. 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, backwards in the current line. A count repeats this search a specified amount of 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.

**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. **N** is the reverse of **n**.

**O** Opens a new line above the current line and inputs text there up to an <*ESC*>. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete as the *slowopen* option works better.

**P** Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines; otherwise, the text is inserted between the characters before and at the cursor. The **P** character may be preceded by a named buffer specification " *x* to retrieve the contents of the buffer. Buffers **1** through **9** contain deleted material, buffers **a** through **z** are available for general use.

**Q** Quits from **vi** to **ex** *command* mode. In this mode, whole lines form commands and end with a <*CR*>. 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). End with an <*ESC*>.

**S** Changes whole lines; a synonym for **cc**. A count substitutes for that many lines. The lines are saved in the numeric buffers and erased on the screen before the substitution begins.

**T** Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as **d**.

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

**V** Unused.

**W** Moves forward to the beginning of a word in the current line where words are defined as sequences of blank/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**<*CR*>). 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 form feed <*CTRL* /*>*. Lines beginning with { also stop [[; this makes it useful for looking backwards, a function at a time, in C programs.

\ Unused.

]] Forwards 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** through **z**), the cursor returns to the position that was marked with this letter. 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 to multiple lines by using <*CR*> 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 ends with an <*ESC*>.

**b** Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics or a sequence of special characters. A count repeats the effect.

**c** An operator that changes the following object, replacing it with the following input text up to an <*ESC*>. If more than part of a single line is affected, the text to be changed is saved in the numeric named buffers. If only part of the current line is affected, the last character to be changed 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 the synonyms (<*CTRL 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 <*CTRL j*> (linefeed) and <*CTRL n*>.

**k** Same as **Up arrow**. Moves the cursor one line up. <*CTRL p*> is a synonym.

**l** Same as **Right arrow**. Moves the cursor one character to the right. **SPACE** is a synonym.

**m** Marks the current position of the cursor in the mark register that is specified by the next character **a** through **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 <*CR*>; 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, this is usually the more useful iteration of **r**.

**s** Changes the single character under the cursor to the text that follows up to an <*ESC*>; given a count, that many characters from the current line are changed. The last character to be changed is marked with **$** as in **c**.

**t** Advances the cursor up to the character before the next character typed. Most useful with 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:

| | |
|---|---|
| <*CR*> | 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 amount 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 vice versa.

*<CTRL ?>* (*<DEL>*)
Interrupts the editor returning it to command-accepting state.