Data General

Customer Documentation

Using the DG/UX™ Editors

AViiON™
PRODUCT LINE

Using the DG/UX™ Editors

069-701036-01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 069-701036

Revision 01, February 1990

NOTICE

Using the DG/UX™ Editors
069-701036-01
069-701065-01 (Japan only)

Revision History:	Effective with:
Original Release – June 1989	DG/UX Release 4.10
Second Release – February 1990	DG/UX Release 4.20

Preface

This manual tells you how to use the editors that come with the DG/UX™ operating system. To use this manual, you should have experience using a computer operating system; while extensive knowledge of a UNIX® system is not necessary for using an editor, you should have some familiarity with the UNIX system. Each of the four editors is distinct from the other, and your application for an editor will determine the one you choose. For instance, **editread** is a command line editor; it allows you to edit commands issued from the shell. Both **vi** and **ed** are text editors; **vi** is a full-screen editor, and **ed** is a line editor. With **vi** you can move the cursor to another location in the file and operate on the desired lines; in **ed**, you issue commands in the command line to affect the file. Finally, **sed** is a batch editor that you use for globally editing multiple files.

Manual Organization

A description of each chapter follows.

Using the Command-Line Editor (editread): Chapter 1
Editread is an optional interface that you can invoke for editing command lines. The **editread** facility can be used with DG/UX system programs.

Using the Full-Screen Editor (vi): Chapter 2
Vi is a full-screen editor that gives you a full range of cursor motion through your file to display the desired text in a window. **Vi** provides tools for appending text, inserting text, opening lines, deleting text, modifying text, and moving text. You can use editing commands to work with characters, words, lines, sentences, and paragraphs.

Using the Line Editor (ed): Chapter 3
Ed is a line editor that is quite versatile because it can be used with any type of terminal including hardcopy terminals. Not only can you use **ed** commands interactively, but you can also use them non-interactively in shell scripts.

Using the Batch Editor (sed): Chapter 4
Sed enables you to perform global editing operations on one or more files. The major difference between the batch editor and the full-screen and line editors is that editing changes with **sed** are written to standard output rather than to the file itself. In this way, you can review dramatic, global changes before altering your source file. Other advantages of **sed** are its speed, its performance of multiple editing commands in one pass, and its use on a pipeline.

Readers, Please Note

Data General manuals use certain symbols and styles of type to indicate different meanings. The Data General symbol and typeface conventions used in this manual are defined in the following list. You should familiarize yourself with these conventions before reading the manual.

This manual also presumes the following meanings for the terms "command line," "format line," and "syntax line." A command line is an example of a command string that you should type verbatim; it is preceded by a system prompt and is followed by a delimiter such as the curved arrow symbol for the New Line key. A format line shows how to structure a command; it shows the variables that must be supplied and the available options. A syntax line is a fragment of program code that shows how to use a particular routine; some syntax lines contain variables.

Convention	Meaning
boldface	In command lines and format lines: Indicates text (including punctuation) that you type verbatim from your keyboard. All DG/UX commands, pathnames, and names of files, directories, and manual pages also use this typeface.
`constant width/ monospace`	Represents a system response on your screen. Syntax lines also use this font.
italic	In format lines: Represents variables for which you supply values; for example, the names of your directories and files, your username and password, and possible arguments to commands.
[*optional*]	In format lines: These brackets surround an optional argument. Don't type the brackets; they only set off what is optional. The brackets are in regular type and should not be confused with the boldface brackets shown below.
[ ]	In format lines: Indicates literal brackets that you should type. These brackets are in boldface type and should not be confused with the regular type brackets shown above.
...	In format lines and syntax lines: Means you can repeat the preceding argument as many times as desired.

$ and %	In command lines and other examples: Represent the system command prompt symbols used for the Bourne and C shells, respectively. Note that your system might use different symbols for the command prompts.
↵	In command lines and other examples: Represents the New Line key, which is the name of the key used to generate a new line. (Note that on some keyboards this key might be called Enter or Return instead of New Line.) Throughout this manual, a space precedes the New Line symbol; this space is used only to improve readability — you can ignore it.
< >	In command lines and other examples: Angle brackets distinguish a command sequence or a keystroke (such as <**Ctrl-D**>, <**Esc**>, and <**3dw**>) from surrounding text. Note that these angle brackets are in regular type and that you do not type them; there are, however, boldface versions of these symbols (described below) that you do type.
<, >, >>	In text, command lines, and other examples: These boldface symbols are redirection operators, used for redirecting input and output. When they appear in boldface type, they are literal characters that you should type.
□	In command lines and other examples: Represents the cursor, which indicates your current typing position on the screen.

Contacting Data General

To order any Data General manual, use the TIPS Order Form at the back of this manual or call a Data General sales representative.

If you have hardware or software problems, please call or write the nearest Data General Office.

If you have comments on this manual, please use the prepaid Customer Documentation Comment Form that appears after the Index of this book. We would appreciate hearing what you like and dislike about this manual.

End of Preface

Contents

Chapter 1 — Using the Command Line Editor: editread

Chapter 2 — Using the Full-Screen Editor: vi

Tables

Figures

Chapter 1
Using the Command Line Editor: editread

The command line editor, called **editread**, is an optional interface that you can invoke for editing command lines that you enter from the shell. Furthermore, **editread** offers a history facility that saves your previously typed commands for later recall and execution. This history facility offers the same basic function as the C shell's history; however, they are different in some important ways, which are discussed in the section "Using History."

NOTE: Programmers—you can also use **editread** with other DG/UX system programs, such as **mxdb** and **crash**. Refer to the *Using the Multi-eXtensible DeBugger (mxdb)* for information on **mxdb**, and the *System Manager's Reference for the DG/UX™ System* for information on **crash**.

What Is History?

The history facility automatically captures and saves a list of the commands you type and execute on a command line. It can save only a certain number of commands, but you can determine that number, up to 500 in a single session. If you find yourself typing repeatedly the same command (or a variation), you could economize on keystrokes by using history. History's recall and repeat facility lets you recall a previous command to re-execute. Additionally, you can recall a command and edit it with **editread** editing keys before re-executing it or you can save the current history as a file and execute the file as a shell script.

Invoking editread for the First Time

The **editread** facility is initially turned off. To enable it and accept the default settings, follow these procedures:

1) In your home directory, create an empty file and name it **.editreadrc**. An example of creating an empty file named **.editreadrc** with the **cat** command follows:

```
$ cat > .editreadrc ⤸
<Ctrl-D>
```

A file named **editreadrc.proto** is provided in **/usr/lib** on your system as a model of a **.editreadrc** file. You may choose to copy this file, renaming it as **.editreadrc**, in your home directory.

2) Verify the existence of the file by listing it:

   $ ls −a .editreadrc ↵

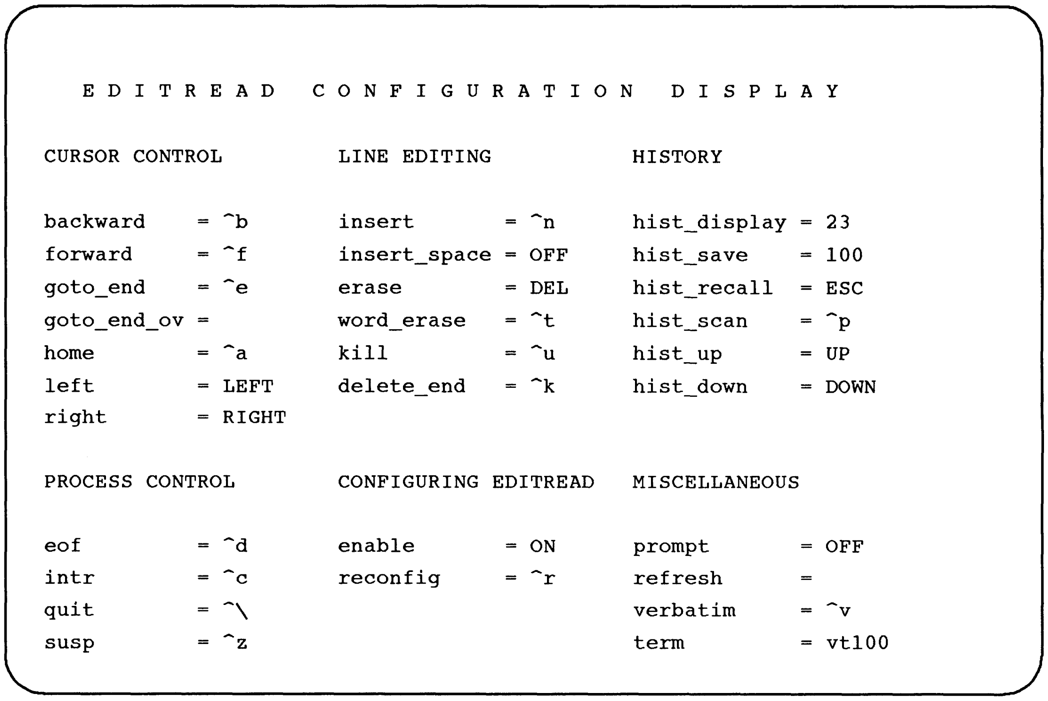
   The **editread** facility is now in effect for new shells or **mxdb** or **crash** processes, but not for the current shell.

Viewing editread's Default Values

To read a quick summary of the **editread** default values, type the following command from the shell:

$ Ctrl-R

Figure 1-1 shows an **editread** configuration display.



```
  E D I T R E A D   C O N F I G U R A T I O N   D I S P L A Y

CURSOR CONTROL          LINE EDITING            HISTORY

backward    = ^b        insert       = ^n       hist_display = 23
forward     = ^f        insert_space = OFF      hist_save    = 100
goto_end    = ^e        erase        = DEL      hist_recall  = ESC
goto_end_ov =           word_erase   = ^t       hist_scan    = ^p
home        = ^a        kill         = ^u       hist_up      = UP
left        = LEFT      delete_end   = ^k       hist_down    = DOWN
right       = RIGHT

PROCESS CONTROL         CONFIGURING EDITREAD    MISCELLANEOUS

eof         = ^d        enable       = ON       prompt       = OFF
intr        = ^c        reconfig     = ^r       refresh      =
quit        = ^\                                verbatim     = ^v
susp        = ^z                                term         = vt100
```

Figure 1-1 Editread Configuration Display

Notice that the caret (^) in this configuration display represents the Ctrl key.

Six categories of functions are identified in the **editread** configuration display:

- Cursor control.
- Line editing.
- History.
- Process control.
- Configuring **editread**.
- Miscellaneous.

Within each category for each function, you will see a keyword that represents the function on the left (such as "backward"); to the right is the current value assigned to the function. A value can be represented as:

- Control key sequence; (Ctrl-*X*), where *X* is another key such as an alphabetic or numeric key.
- Single key; such as "RIGHT," "LEFT," "UP," and "DOWN," which correspond to the arrow keys: rightarrow (→), leftarrow (←), uparrow (↑), and downarrow (↓) on the cursor motion keypad. Also, the at sign (@) and the pound sign (#) are single keys.
- String value; such as the type of terminal you are using.
- Number value; such as history display length.
- ON or OFF condition; ON enables a function, and OFF disables a function. An unassigned function is considered off.
- No setting means that the function is disabled (or unassigned), but you can assign a key sequence or single key value to it. To disable a function, assign the OFF value.

Table 1-1 summarizes the **editread** functions and default values:

NOTE: In the following table, the Ctrl key is used to represent the caret (^).

Table 1-1 Summary of editread Functions and Values

Keyword	Description	Default
	Editread Configuration	
enable	Turns **editread** off or on.	ON
reconfig	Displays the current **editread** values and concludes a redefinition of a single function or multiple functions.	Ctrl-R
	Cursor Control	
right	Moves the cursor one position to the right.	rightarrow
left	Moves the cursor one position to the left.	leftarrow
goto_end	Moves the cursor to the end of the line.	Ctrl-E
goto_end_ov	Recalls the previous line and moves the cursor to the end of it.	unassigned
home	Returns the cursor to the beginning of the line.	Ctrl-A
forward	Moves the cursor to the beginning of the next word.	Ctrl-F
backward	Moves the cursor backward to the space after the previous word.	Ctrl-B
	Line Editing	
erase	Erases a character one position to the left of the cursor.	Del
word_erase	Deletes the current word.	Ctrl-T
delete_end	Deletes from the cursor position to the end of the line.	Ctrl-K
kill	Erases the entire line.	Ctrl-U
insert	Enables and disables insert mode.	Ctrl-N
insert_space	When insert mode is in effect, a leading space always appears when inserting characters to the left of the cursor.	OFF
	History	
hist_recall	Displays the history list.	Esc
hist_scan	Searches through history list for pattern matches.	Ctrl-P
hist_up	Recalls the previous command in history.	uparrow
hist_down	Recalls the next command in the history list.	downarrow
hist_save	Sets the maximum number of commands to be saved in history.	100
hist_display	Sets the number of commands to be displayed at one time on your screen when you press **<Esc>**.	23

(continued)

Table 1-1 Summary of editread Functions and Values

Keyword	Description	Default
	History (continued)	
write_hist	Is not a function you can configure, but a command you can use to write history commands to a file.	**write_hist** *file*<**Ctrl-R**>
read_hist	Is not a function you can configure, but a command you can use to read a file containing a history list into the current history list.	**read_hist** *file*<**Ctrl-R**>
	Process Control	
eof	Sets the end-of-file character.	Ctrl-D
intr	Sets the interrupt key.	Del
quit	Sets the quit key.	Ctrl-\
susp	Sets the suspend key.	Ctrl-Z
	Miscellaneous	
prompt	Precedes the shell prompt symbol with the current history number.	OFF
refresh	Refreshes the current line.	unassigned
verbatim	Nullifies (escapes) the meaning of an **editread** value.	Ctrl-V
term	Identifies your terminal type.	vt100

(concluded)

Assigning Values to editread Functions

You can assign values to functions in these places:

- The current program (**sh**, **csh**, **mxdb**, or **crash**).
- The **EDITREAD** environment variable.
- The **.editreadrc** file in your current directory.
- The **.editreadrc** file in your home directory.

If any **editread** function is assigned interactively in the current program, it is in effect only for the current program; it is not exported to child processes or other shells. Also, when you log out of the system, the assigned functions are dismissed. (See the next section for information on assigning values interactively.)

Each time you log in to your system or create a new shell, the system retrieves **editread** values using this initialization sequence:

1) If the **EDITREAD** environment variable is set, it is used to enable **editread** in new programs that are created. This environment variable is in effect only for the current session (until you log out of the system), unless you choose to

set the **EDITREAD** environment variable in the appropriate setup file: **.profile** for the Bourne shell, and **.login** for the C shell. If the **EDITREAD** environment variable exists, the **.editreadrc** files in your current directory and your home directory are ignored.

2) If no **EDITREAD** variable is set, but you have a **.editreadrc** file in your current directory, the file will be used to enable **editread** each time you log in to the system or create a new shell. If you have a **.editreadrc** file in your home directory, it will be ignored.

3) If there is no **.editreadrc** file in your current directory, then the **.editreadrc** file in your home directory is used to enable **editread**.

Changing editread Values Interactively

When you change **editread** values interactively, the changed values will be in effect only until you leave the program or you log out of the system. The default values from your **.editreadrc** file or the **EDITREAD** environment variable will be restored automatically for subsequent sessions. An advantage to setting a value interactively is that the change becomes effective immediately without your having to start a new process (logging out is one way to start a new shell), which is required when you assign values in a **.editreadrc** file or to the **EDITREAD** environment variable. A value set interactively overrides the same value set either in a **.editreadrc** file or the **EDITREAD** environment variable.

You can reconfigure **editread** functions interactively using this command format:

function keyword= *value* [*function keyword*=*value*]**<Ctrl-R>**

You can make single or multiple function assignments per line, which is terminated by Ctrl-R. Examples of assigning new values to the functions follow:

```
$ erase = ^?<Ctrl-R>
$ intr=^c hist_recall=^z  prompt=OFF<Ctrl-R>
$    hist_display  =  12   reconfig=f1<Ctrl-R>
```

In the first line, the erase character function is assigned to Ctrl-?, which is the Del key. In the second line, the interrupt function is assigned to Ctrl-C, history recall to Ctrl-Z, and the history prompt is off. In the last line, the history display depth is set to 12 commands and the reconfiguration key is changed from Ctrl-R to the F1 function key. Any number of spaces can be inserted for readability. At least one space must separate a value from the next function, however. In this example, several spaces precede the function name on the line.

If one function has two values, for example,

```
erase = ^?
erase = ^e
```

the last assignment will take effect, overwriting the previous assignment. To find out whether your assignment took effect, view the current configuration settings with the Ctrl-R command and to try out the function.

If two functions have the same value, for example,

```
erase = ^?
delete_end = ^?
```

only the first assignment will take effect, and the alternate assignment will not. No error will be indicated. Only trial and error will show which one is successful.

Assigning Values to the EDITREAD Environment Variable

From the shell, you can assign values to the environment variable **EDITREAD**. Examples follow:

Bourne shell:

```
$ EDITREAD='prompt = ON  goto_end = OFF  goto_end_ov = ^a' ↵
$ export EDITREAD ↵
```

C shell:

```
% setenv EDITREAD 'prompt = ON  goto_end = OFF  goto_end_ov = ^a' ↵
```

NOTE: For Bourne Shell Users, no spaces can separate the equal sign from the single quotation mark and the **EDITREAD** variable name. Your function assignments can occupy multiple lines; just be sure to terminate each line by pressing the New Line key. Regardless of the number of lines occupied, the assignments are enclosed by a pair of single quotation marks (' '). The closing quotation mark terminates your input.

To verify the current values assigned to the **EDITREAD** environment variable, type this command:

```
$ env ↵
```

The current values will be displayed.

Assigning Values in the .editreadrc File

Using an editor or the **cat** command, you can edit the **.editreadrc** file in your home directory. An example of how to assign values to **editread** functions in the **.editreadrc** file follows:

```
$ cat > .editreadrc ↵
erase = ^? ↵
intr = ^c  hist_recall = Esc  prompt = ON ↵
hist_disp=12    reconfig=f1 ↵
<Ctrl-D>
```

You must end each line in the **.editreadrc** file by pressing the New Line key. You can edit the **.editreadrc** file using any editor. The functions that you do not specify will assume the default values (which you can see by using the Ctrl-R command for the display of the current configuration). To make your changes take effect, you will have to log out of, then back in to the system.

Your Line Discipline and editread

By default, the line-editing and control keys defined in the **editread** facility are copied from your terminal's line discipline. As an alternative, you may choose to redefine some or all keys in your **.profile** file (or **.login** file for C shell users) and those key definitions will be exported to **editread**. As an example, **editread** and your terminal are set up to use Del as the erase key and Ctrl-U as the delete line key. If you redefine a function in the **editread** facility, you should make a corresponding change to your line discipline. **Editread** and your line discipline should be in agreement. Refer to *Using the DG/UX™ System* for more information on setting your line discipline.

Disabling editread

By default, **editread** is enabled, or on. This function is useful if you choose to disable **editread** either temporarily during a session or permanently. For example, you may want to use **editread** when running specific programs in the Bourne shell; however, you may prefer to use the C shell without **editread**.

The syntax for changing this function follows:

```
enable=value<Ctrl-R>
```

An example follows:

```
$ enable=OFF<Ctrl-R>
```

This example disables **editread** for the current shell.

```
% enable=OFF<Ctrl-R>
```

To disable **editread** for all subshells and other programs, you can assign the OFF value to the **enable** function in the **EDITREAD** environment variable. An example follows:

Bourne shell:

```
$ EDITREAD='enable=off' ↵
$ export EDITREAD ↵
```

C shell:

```
% setenv EDITREAD 'enable=OFF' ↵
```

To disable **editread** permanently, you would have to delete the **.editreadrc** file and remove the **EDITREAD** environment variable, or set **enable=off** in the **.editreadrc** file or in the **EDITREAD** environment variable in the appropriate setup file, which is **.login** for C shell users, and **.profile** for Bourne shell users.

Defining Cursor Control Keys

The cursor control functions are:

- Moving the cursor right.
- Moving the cursor left.
- Moving the cursor to the end of the line.
- Retrieving the previous line and putting the cursor at the end.
- Returning the cursor to the beginning of the line.
- Moving the cursor forward one word.
- Moving the cursor backward one word.

Moving the Cursor Right

By default, you can use the rightarrow key (→) in the cursor motion keypad to advance the cursor right. This is a nondestructive right move; you can move the cursor over existing text without erasing it. If the cursor reaches the end of the line, the cursor remains stationary and the terminal beeps.

The syntax for changing this function follows:

right=*value***<Ctrl-R>**

An example follows:

```
$ right=^U<Ctrl-R>
```

This example reassigns Ctrl-U to the move-cursor-right function.

Moving the Cursor Left

By default, you can use the leftarrow key (←) in the cursor motion keypad to back up the cursor. This is a nondestructive left move; you can move the cursor over existing text without erasing it. If the cursor reaches the beginning of the line, the cursor remains stationary and the terminal beeps.

The syntax for changing this function follows:

left=*value***<Ctrl-R>**

An example follows:

```
$ left=^U<Ctrl-R>
```

This example reassigns Ctrl-U to the move-cursor-left function.

Moving the Cursor to the End of the Line

By default, you can use the Ctrl-E key to cursor to the end of the line, the position following the final character.

The syntax for changing this function follows:

goto_end=*value***<Ctrl-R>**

An example follows:

```
$ goto_end=^U<Ctrl-R>
```

This example reassigns Ctrl-U to the goto_end function.

Retrieving the Previous Line and Putting the Cursor at the End

This function appends the rest of the previous event from history and puts the cursor at the end of the line. By default, this function is unassigned; to enable it, you can assign the function to a desired key.

The syntax for changing this function follows:

```
goto_end_ov=value<Ctrl-R>
```

An example follows:

```
$ goto_end_ov=^U<Ctrl-R>
```

This example reassigns Ctrl-U to the goto_end_ov function. The following example shows the behavior of goto_end_ov.

```
$ cd /usr/della ↵
$ ls<Ctrl-G>
ls /usr/della□
```

Returning the Cursor to the Beginning of the Line

By default, you can use the Ctrl-A key to move the cursor to the first position in the line.

The syntax for changing this function follows:

```
home=value<Ctrl-R>
```

An example follows:

```
$ home=^U<Ctrl-R>
```

This example reassigns Ctrl-U to the home function.

Moving the Cursor Forward One Word

By default, you can use the Ctrl-F key to move the cursor forward to the first character of the next word. If there is no text on the current line after the cursor when you press Ctrl-F, the corresponding word (the word at the cursor position in the previous line) is recalled to the screen, and the cursor is advanced to the end of that word in the line. A word is one or more characters surrounded by white space.

The syntax for changing this function follows:

forward=*value***<Ctrl-R>**

An example follows:

```
$ forward=^U<Ctrl-R>
```

This example reassigns Ctrl-U to the forward function.

Moving the Cursor Backward One Word

By default, you can use the Ctrl-B key to move the cursor backward to the space after the last character in the previous word. If there is no remaining text on the screen, the cursor goes to the first position. If the cursor is already in the first position of a line, the terminal beeps.

The syntax for changing this function follows:

backward=*value***<Ctrl-R>**

An example follows:

```
$ backward=^U<Ctrl-R>
```

This example reassigns Ctrl-U to the backward function.

Defining Line Editing Keys

The line-editing functions are:

- Erasing a character backward.
- Erasing a word.
- Deleting from the cursor to the end of the line.
- Killing (deleting) an entire line.
- Inserting text within a line.
- Inserting text with a leading space.

Erasing a Character Backward

By default, you can use the Del key to erase the character one position to the left of the cursor. If you press this key repeatedly, characters to the left of the cursor will be erased continuously in a backward direction. If the cursor reaches the beginning of the line, the cursor remains stationary and the terminal beeps.

The syntax for changing this function follows:

erase=*value***<Ctrl-R>**

An example follows:

$ **erase=^U<Ctrl-R>**

This example reassigns Ctrl-U to the erase function.

Erasing a Word

By default, you can use the Ctrl-T key to erase a word or partial word starting at the cursor position through the end of the word.

The syntax for changing this function follows:

work_erase=*value***<Ctrl-R>**

An example follows:

$ **word_erase=^U<Ctrl-R>**

This example reassigns Ctrl-U to the word_erase function.

Deleting from the Cursor to the End of the Line

By default, you can use the Ctrl-K to delete all text from the current cursor position to the end of the line.

The syntax for changing this function follows:

delete_end=*value***<Ctrl-R>**

An example follows:

$ **delete_end=^U<Ctrl-R>**

This example reassigns Ctrl-U to the delete_end function.

Deleting an Entire Line

By default, you can use the Ctrl-U key to erase the entire line. The kill character for **editread** is the same kill character in the shell. The kill characters for **editread** and the shell should be the same.

The syntax for changing this function follows:

kill=*value***<Ctrl-R>**

An example follows:

```
$ kill=^H<Ctrl-R>
```

This example reassigns Ctrl-H to the kill function.

Inserting Text Within a Line

By default, you can use the Ctrl-N key to turn on insert mode. To edit a line using insert, you can press Ctrl-N at the desired position to begin inserting text. Characters you type are inserted to the left of the cursor. When you finish inserting text, press Ctrl-N again to turn off insert mode and continue normal text entry. Other **editread** functions (such as left, right, or home) are also in effect in insert mode. You can move the cursor within the line and insert text where desired.

The syntax for changing this function follows:

insert=*value***<Ctrl-R>**

An example follows:

```
$ insert=^U<Ctrl-R>
```

This example reassigns Ctrl-U to the insert function.

Inserting Text With a Leading Space

By default, the insert_space function is off. This function is related to the insert mode function (refer to the previous section on insert mode). Insert mode must be active before this function will be effective. When both functions are on, a leading space will precede the characters that you type to the left of the cursor. The leading space is removed from the line when you deactivate insert mode with Ctrl-N. The function is off by default; to enable it, turn it on. If you are using a slow terminal, this function may slow down your system response.

The syntax for changing this function follows:

```
insert_space=value<Ctrl-R>
```

An example follows:

```
$ insert_space=ON<Ctrl-R>
```

This example enables the insert_space function.

Defining History Keys

A major feature of **editread** is its history facility, which may be particularly useful for Bourne shell users. If, however, you are a C shell user, you may prefer to use the history facility in **editread** rather than in the C shell.

Differences Between editread and C Shell History

The major differences between the **editread** and C shell history facilities is the method of recalling and editing history events. (Refer to *Using the DG/UX™ System* for more information on the C shell history facility.)

In the **editread** facility, you can use the arrow keys to scan up and down through the history list to retrieve the desired item. In the C shell, you can invoke an event using an exclamation point (!) and the item's absolute or relative number or a unique pattern.

In the **editread** facility, you can edit events using the editing keys. In the C shell, you can use a set of editing commands to access specific items in the event and perform substitutions.

The **editread** history functions are:

- Recalling a history event.
- Scanning the history list.
- Moving up the history list.
- Moving down the history list.
- Setting the maximum history length.
- Setting the history display length.
- Writing the history list to a file.

- Reading a file containing a current history list.

Recalling a History Event

The history recall function searches for the history event number or the most recent history event that matches the regular expression that you provide and places it on the command line as the current event. You initiate the search by pressing the Escape key. If there is no match, your terminal beeps. If you press the Escape key with no argument, the history list (whose length is defined by the history display length function covered in a later section) is displayed. You use regular expression syntax for setting up search patterns. Refer to Appendix A in this manual for information on regular-expression pattern matching.

When you recall a history event and edit it using the line-editing commands, you will be editing the current item instead of changing history. Once you terminate a current item with a New Line, the item is entered in history and will not change.

The syntax for this function follows:

hist_recall=*value***<Ctrl-R>**

An example follows:

$ **hist_recall=r<Ctrl-R>**

This example reassigns the **r** key to the hist_recall function.

The history list that you recall with the Escape key follows:

```
$ <Esc>
     12 pwd
     13 ls -l
     14 cd bin
     15 cat test
     16 ls test
     17 cat rdiffmark
     18 vi rdiffmark
     19 cp delblanks rdiffmark
     20 rm rdiffmark
$ 21$
```

The history prompt has been enabled for the display of this history list (refer to the section, "Displaying the History Prompt," later in this chapter for more information).

From the command line, you can issue the following command to recall a previous event from the history list:

```
$ 21$ cp<Esc>
     19 cp delblanks rdiffmark
$ 21$
```

Using regular-expression pattern matching, you request the display of the most recent command in the history list that begins with **cp**. You terminate a history recall command with the Escape key. Event 19 is recalled as the current event.

Scanning the History List

By default, you can use the Ctrl-P key to terminate a regular expression when scanning a history list. The history scan function is similar to the history recall function, except that all pattern matches (rather than just the most recent match) and associated prompt number (if the prompt function is on) are displayed. In the following example, using regular-expression pattern matching, you request the display of all commands in the history list that begin (^) with a **c**. You terminate this command with Ctrl-P.

The syntax for changing this function follows:

hist_scan=*value***<Ctrl-R>**

An example follows:

```
$ hist_scan=^U<Ctrl-R>
```

This example reassigns Ctrl-U to the hist_scan function.

An example of its default use follows.

```
$21 ^c<Ctrl-P>
     19 cp delblanks rdiffmark
     17 cat rdiffmark
     15 cat test
     14 cd bin
$21
```

All of these events begin with the letter **c**.

Moving Up the History List

By default, you can use the uparrow (↑) key to move up the history list. This command recalls the previous command from the history list and displays it as the current line. Each successive press of the uparrow key causes the previous command from the history list to be displayed on the command line. You can cycle through the history list (from most to least recent command) by repeatedly pressing the uparrow key. When you reach the oldest command in the history list, pressing the uparrow key again will display the current command line (the most recent command).

The syntax for changing this function follows:

hist_up=*value***<Ctrl-R>**

An example follows:

$ **hist_up=^U<Ctrl-R>**

This example reassigns Ctrl-U to the hist_up function.

Moving Down the History List

By default, you can use the downarrow (↓) key to move down the history list. It recalls the next command from the history list and displays it as the current line. Each successive press of the downarrow key causes the next command in the history list to be displayed on the command line. You can cycle through the history list (from least to most recent command) by repeatedly pressing the downarrow key. When you reach the most recent command in the history list, pressing the downarrow key again will repeat the search through the list starting at the oldest command.

The syntax for changing this function follows:

hist_down=*value***<Ctrl-R>**

An example follows:

$ **hist_down=^U<Ctrl-R>**

This example reassigns Ctrl-U to the hist_down function.

Setting the Maximum History Length

The history save function sets the maximum number of history items to be saved during a single session (while you are logged in). The minimum value is 1; the maximum 500. This function is assigned the value 100 by default. To change the value, assign the function to a legal number. You should select an economical history length because the longer the list, the slower your system will respond when displaying history.

If you set your history length to 15, as you type your 16th command on the command line, it is recorded as event 16 in the history list and event 1 is deleted; thus a constant list of 15 events is maintained.

The syntax for changing this function follows:

hist_save=*value***<Ctrl-R>**

An example follows:

```
$ hist_save=15<Ctrl-R>
```

This example sets the hist_save function to 15.

Setting the History Display Length

By default, the history display length is 23. Regardless of the maximum number of history items saved, you can choose the number of commands in history to be displayed each time you press the Escape key. The minimum value is 0; the maximum is the value of the history length function (see the previous section).

You should select a history display length that does not exceed the size of your screen. If your history display length exceeds a screen, the remaining portion will scroll up and off the screen.

The syntax for changing this function follows:

hist_display=*value***<Ctrl-R>**

An example follows:

```
$ hist_display=10<Ctrl-R>
```

This example sets the hist_display function to 10.

Writing the History List to a File

The **write_hist** command writes the entire history list to a file, which you can use as you wish. For example, if you are writing a shell script interactively, after you have debugged the commands, you may want to save them in a file that you can run as an executable shell script.

The syntax follows:

write_hist *filename***<Ctrl-R>**

An example follows:

```
$ write_hist historyfile<Ctrl-R>
```

This example writes the current history list to a file named **historyfile**.

Reading a File Containing a History List

Often, you will use the same set of commands when you start **mxdb** or **crash**. It is easier to save a set of commands in a file, using the **write_hist** command (previous section), and then read them in for the next session using the **read_hist** command. With a history list intact, you can then retrieve or edit the desired command for reexecution without having to retype the commands from scratch.

This command reads a file containing command lines (which you can create with an editor or by writing the history list to a file) and appends it to the current history list. These commands are not executed, they are just appended to history.

To confirm a successful read operation, you can press the Escape key and see your updated history list displayed on the screen.

If you inadvertently name an incorrect file to be read, an error message is displayed.

The syntax for the **read_hist** command follows.

read_hist *filename*<**Ctrl-R**>

An example follows:

```
$ read_hist historyfile<Ctrl-R>
```

This example causes the file named **historyfile** to be read into the current history list.

Defining Process Control Keys

The process control commands are designed to interrupt the execution of a process. The process control keys are:

- End-of-file (such as to log out or to end input).
- Interrupt (terminate a process).
- Quit.
- Suspend.

Each of the process control functions is set to your terminal's settings by default. The settings you make through **editread** should be consistent with those set by your terminal. If there is a discrepancy, while **editread** is in effect, the values you set in **editread** will override the terminal default settings. Consult your system administrator for information on default terminal settings at your site.

The End-of-File Key (Log Out and End Input)

By default, you can use the Ctrl-D key to set the end-of-file (log out or end input) control character.

The syntax for changing this function follows:

eof=*value***<Ctrl-R>**

An example follows:

$ **eof=^U<Ctrl-R>**

This example reassigns Ctrl-U to the eof function.

The Interrupt Key

You can use the Ctrl-C key to set the interrupt key, which terminates a process.

The syntax for changing this function follows:

intr=*value***<Ctrl-R>**

An example follows:

$ **intr=^U<Ctrl-R>**

This example reassigns Ctrl-U to the intr function.

The Quit Key

By default, you can use the Ctrl-\ key to set the quit signal key, which terminates the process but saves an image of memory in a file named **core** in your current directory, which can be analyzed.

The syntax for changing this function follows:

quit=*value***<Ctrl-R>**

An example follows:

$ **quit=^U<Ctrl-R>**

This example reassigns Ctrl-U to the quit function.

The Suspend Key

The suspend function is useful for interrupting a normal program process. By default, this function is unassigned in the Bourne shell but is set in the C shell. It is useful for suspending processes (refer to *Using the DG/UX™ System* for this information).

The syntax for changing this function follows:

susp=*value***<Ctrl-R>**

An example follows:

$ **susp=^U<Ctrl-R>**

This example reassigns Ctrl-U to the susp function.

Defining Miscellaneous Keys

This group contains four functions not categorized elsewhere. These functions are:

- Displaying the history prompt.
- Refreshing the current command line display.
- Enabling verbatim mode.
- Displaying the terminal type.

Displaying the History Prompt

This function is turned off by default; to enable it, assign it the on value. Each command you enter from the shell or other program is automatically preceded by a history prompt number, letting you know the current history number of the command you are entering.

The syntax for changing this function follows:

prompt=*value***<Ctrl-R>**

An example follows:

$ **prompt=ON<Ctrl-R>**

This example enables the prompt function.

An example of the display of the history number follows:

```
$ 32$
```

This function is particularly useful for scanning the history list. The desired command and its corresponding history list number are echoed to the command line each time you press a scan key (the ↑ or ↓ keys, by default) to move up or down the history list.

Refreshing the Current Command Line Display

By default, the refresh function is unassigned. If your screen becomes littered with spurious marks, which can result from modem "noise" or broadcast messages, you can redraw the current command line on the line below the one you were working on by using the key to which you assign the refresh function.

The syntax for changing this function follows:

refresh=*value***<Ctrl-R>**

An example follows:

$ **refresh=^U<Ctrl-R>**

This example reassigns Ctrl-U to the refresh function.

Enabling Verbatim Mode

By default, you can use the Ctrl-V key to escape the interpreted meaning of an expression, which presents the literal expression on the screen. For example, if you type Ctrl-T at the shell prompt, it is interpreted as an **editread** command to delete a word. If, however, you want to type the literal expression **^t**, without interpretation by **editread**, you must precede it with the keystroke Ctrl-V.

The syntax for changing this function follows:

verbatim=*value***<Ctrl-R>**

An example follows:

$ **verbatim=^U<Ctrl-R>**

This example reassigns Ctrl-U to the verbatim function.

An example of using the default function follows.

<Ctrl-U> <Ctrl-T>

You will see:

```
^t
```

Displaying Terminal Type

Your terminal type is typically defined in your setup file; **.profile** for the Bourne shell, and **.login** for the C shell. You assign your terminal type to the environment variable **TERM**. Regardless of where your terminal is defined, that assignment must also be reflected in **editread**. The easiest way to check your current terminal type is to enter this command:

```
$ echo $TERM ↵
vt100
```

In this example, **vt100** was reported as the terminal type. If you have any questions about your current terminal type, check with your system administrator.

The syntax for changing this function follows:

term=*value***<Ctrl-R>**

An example follows:

```
$ term=d215<Ctrl-R>
```

This example reassigns **d215** to the term function.

End of Chapter

Chapter 2
Using the Full-Screen Editor: vi

Vi, pronounced "vee-eye," is an interactive, visually oriented full-screen editor. Your terminal screen behaves as a window through which you view a portion of a file. The **vi** editor has commands to move the cursor in recognized units (such as a character, word, line, sentence, and paragraph) and to perform various editing operations. A combination of the cursor movement and editing commands lets you identify the text for display in the current window and define the text for editing (such as deleting several sentences or changing several words). This regularity in command syntax helps you to remember and apply the commands.

The first several sections of this chapter (through "Vi Editing Operations") give an orientation to **vi**; the rest of the chapter gives a detailed description of the **vi** editing operations and provides examples of user input and the results produced.

This chapter is not intended as a tutorial; however, you may want to try out the examples as you go. If you already know the DG/UX system or another UNIX system, you may choose to read carefully the first part of the chapter up through "Vi Editing Operations," and then seek out topics that interest you.

The Working Buffer

When you type text into a file or edit an existing file, you are actually working on a copy of the original disk file that is placed in a temporary workspace called the working buffer. The permanent file remains unchanged until you issue a command to overwrite the permanent disk file with the contents of the working buffer. When you create a new file, **vi** will not create it until you actually issue a command to write the working buffer to the file.

If your computer system crashes, the contents of your buffer space may be lost (there are, however, some ways to recover from errors). To prevent this occurrence, you are advised to make periodic saves of your working buffer. Alternatively, you may choose to dismiss all changes made to the current working buffer, thereby keeping the original disk file intact. To perform periodic saves or to dismiss the contents of the work buffer, you can issue specific commands through last line mode, which is discussed in the next section.

How vi Operates

Vi operates in three modes:

- Input.
- Command.
- Last line.

Input Mode and Command Modes

While in input mode, **vi** accepts your input as text, which is displayed as you type it. To perform any operation besides entering text (such as moving the cursor and making editing corrections), you have to change **vi** to command mode. While in command mode, **vi** interprets your input as commands, executing each command as you type it. Your commands are not displayed on the screen.

When you begin an editing session in **vi**, command mode is the default. Start **vi** using the following command:

```
$ vi test ⮐
```

The first key you press will be interpreted as a command. To begin entering text, you must first issue a command that allows you to enter text. The command is not displayed. Enter this command to append text to a new file:

```
a
```

Then, without pressing the space bar first, enter the text, shown as follows.

```
The night was dark and dreary ⮐
when I decided to sit down and ⮐
write a letter to my friend.□
```

Finally, use the Escape key to change from input mode to command mode.

In the previous example, you invoked **vi**, named file **test**, and used the **a** command to indicate that you wanted to append (or enter) text in a file. Note that the **a** command is not displayed on the screen. Then, you typed three lines of text, the first two lines are terminated by pressing the New Line key. Finally, you pressed the Escape key to end input mode to return to command mode. When you pressed the Escape key, the cursor moved back a column position, over the period.

If you press the Escape key while in command mode, your terminal will beep in response. On some terminals, the screen will flash rather than beep. Pressing the Escape key repeatedly does not affect your file.

Last Line Mode

A third mode offered in **vi** is last line mode, which provides an extended set of editing commands. In command mode, you can escape to last line mode temporarily by typing a colon (**:**). The character you type will be echoed as a prompt on the bottom line of the screen, typically line 24. The colon means that the command you type will affect the file in a forward direction. In addition, you can escape to last line mode using a question mark (**?**), which means that the command you type will affect the file in a backward direction.

The following four sections in this manual cover the types of functions you can perform from last line mode:

- Searching for patterns in last line mode
- Manipulating files in last line mode
- Setting up your vi environment
- Writing and using macros

Assuming that **vi** is in command mode (you've already pressed the Escape key), let's say that you want to search for the literal pattern, "dreary," in the following text.

```
The night was dark and dreary
when I decided to sit down and
write a letter to my friend.
```

From the cursor position, you want to search in a backward direction, which you signify with the question mark (?), for the word, "dreary." You can type **?** without first preceding it with **:**, which signifies last line mode. You end the command by pressing the New Line key.

?dreary ↵

The cursor will be positioned automatically on the first character of the pattern, as follows.

```
The night was dark and dreary
when I decided to sit down and
write a letter to my friend.
```

Refer to the later section "Quitting vi" for instructions on closing a file.

Defining a Terminal for vi

Vi is designed to operate with a variety of terminal types; therefore, before using **vi**, you must tell **vi** the type of terminal you have. Your system administrator probably will have taken care of this matter for you. Your terminal is defined in the **TERM** environment variable or in your setup file—**.profile** for Bourne shell users and **.login** for C shell users. The appropriate setup file must be located in your home directory. Refer to Appendix A in *Using the DG/UX™ System* for information on setting up your terminal.

Differences Among Terminals Used with vi

The editing examples in this chapter assume certain keyboard conventions such as the use of the New Line key and the Del (Delete) key. Some terminals have a CR (Carriage Return) key instead of a New Line key. On these terminals, you should use the Carriage Return. If these keys don't work, you may need to check your line discipline, which is discussed in *Using the DG/UX™ System*. Consult with your system administrator if you have problems.

Also, this chapter assumes that your terminal provides "new" operations such as insert and delete characters and lines, which allow **vi** to keep the screen completely up-to-date. The examples of terminal output in this chapter may appear somewhat different from the results produced on your terminal screen. A few differences are as follows.

- On some terminals, text that you append and insert may appear to overwrite existing text. The existing text will reappear as soon as you press the Escape key to return **vi** to command mode. Setting the redraw option (in last line mode) will cause your editing action to take effect immediately, thus preventing the appearance of overwriting. Setting the redraw option is covered in a later section "Setting Up Your vi Environment."

- Another trait of some terminals is the appearance of at (@) signs in the first column of the screen that results from line deletions. Setting the redraw option also prevents the display of delete symbols. If, however, you do not set the redraw option, to get rid of these symbols and close up the remaining lines, you can type Ctrl-R, which refreshes the screen.

- Furthermore, on some terminals, pressing the Escape key repeatedly causes your terminal to beep. The same action on other terminals will cause a brief flash (this is a visual beep) if you set the flash option. Neither response is necessarily considered an error.

The Syntax of a vi Command

To perform an editing function from **vi** command mode, you issue a short command, which defines precisely the operation to be performed. Each command follows this syntax:

[*number*]*editing-command*[*text-object*]

where:

number is the number of text objects you wish to operate on; using a number depends on the command being performed.

editing-command causes a specific operation to be performed on the text, such as to append, delete, or change text. Table 2-1 identifies the editing operations performed on text objects:

Table 2-1 Vi Editing Operations

Editing Operation	Description
Scroll full screen forward	Moves the display one full screen forward (or up).
Scroll full screen backward	Moves the display one full screen backward (or down).
Scroll half-screen up	Moves the display one-half screen up (or forward).
Scroll half-screen down	Moves the display one-half screen down (or backward).
Append	Adds text to the right of the cursor.
Insert	Allows text to be inserted to the left of the cursor.
Open line	Inserts a blank line at the cursor position.
Delete	Erases a pre-defined range of text to the right or left of the cursor.
Replace	Overwrites text with new text to the right of the cursor.
Change	Substitutes a pre-defined text string for a pre-defined range of text to the right or left of the cursor.
Yank (move)	Moves a specified range of text to the right or left of the cursor.
Put	Places a specified range of text to the right or left of the cursor.
Mark	Delimits a boundary of text to be yanked, changed, or deleted.

Each of these editing operations is described thoroughly in this chapter.

text-object is the range of text you want to operate on; using a text object depends on the command being performed.

NOTE: No spaces occur between command items. Also, commands are case sensitive; you should use the command in the exact case it is presented in. Whether or not to use a *number* or a *text-object* depends on the particular command. Each command is terminated by pressing a New Line (or Carriage Return) key.

Text Objects

When you work on files containing large bodies of text, it may be advantageous to treat text as units (such as a word, sentence, or paragraph) rather than individual characters. Vi recognizes a set of text objects that you can use to identify the specific text on which an editing function will operate. They are:

- Character.
- Word.
- Space-delimited word.
- Line.
- Sentence.
- Paragraph.
- Screen window.
- Mark.

Character

A character is a single byte — a letter, number, punctuation mark, or a nondisplayable control sequence such as a tab (represented either by pressing the Tab key or Ctrl-I). Examples of characters follow:

```
a    A
0    9
.    ?
*    \
Tab  space bar
```

Word

A word is a group of adjacent characters bounded on both sides by any combination of these items: punctuation, space, tab, number, or a new-line. A single punctuation mark counts as a single word; multiple punctuation marks (with no intervening space) also count as a single word. Examples of words follow:

Words	Number of Words
`man`	1
`man.`	2
`man).`	2
`man). He`	3
`man). He "`	4
`man). He said, "`	6
`(a young man).  He said, "How are you, son?"`	15

Space-Delimited Word

A space-delimited word is the same as a word except that it includes adjacent punctuation. Space-delimited words are separated by one or more of these items: space, tab, or new-line. Examples of space-delimited word follow:

Space-Delimited Word	Number of Space-Delimited Words
`man`	1
`man.`	1
`man).`	1
`man). He`	2
`man). He "`	3
`man). He said, "`	4
`(a young man).  He said, "How are you, son?"`	9

Line

A line is a group of characters including spaces and punctuation that ends with a new-line. A line is not necessarily a single, physical line on your terminal screen. A line can span several physical lines by wrapping. Only a deliberate press of the New Line key will terminate a line. Examples of lines follow:

Lines	**Total Number of Lines**

`Paychecks are here! ↵`
`Where are they?  Jackson ↵`
`Page, noted for his`
`industry and ambition,`
`was awarded the "Most Valuable`
`Staffer" award on Monday, ↵`
	3

The preceding example shows a total of three lines.

Sentence

A sentence is a group of characters ending with final punctuation and two spaces. Final punctuation includes a period (.), question mark (?), and exclamation point (!). If a sentence coincidentally ends at the end of a line (no continuation to the next line), the new-line following punctuation implies a sentence. Spacing twice is not necessary in this case. Examples of sentences follow:

Sentence	**Number of Sentences**
`Paychecks are here! Where are they? ↵`	1
`Paychecks are here!  Where are they? ↵`	2
`Paychecks?  Where?  I want mine. ↵`	3

`Jackson Page, noted for his industry`
`and ambition, was awarded the "Most`
`Valuable Staffer" award on Monday,`
	1

Paragraph

A paragraph is a group of characters bounded on both sides by one or more blank lines. Examples of paragraphs follow:

```
Paragraphs                                               Number of
                                                         Paragraphs


Paychecks are here!                                         1


     Paychecks are here!                                    1
Where are they?
     Jackson Page, noted for
his industry and ambition


Paychecks are here!                                         3

Where are they?

Jackson Page, noted for his
industry and ambition
```

Screen Window

The screen is defined as the text displayed within the current window. The default number of lines per screen is one less than the number of lines on your terminal display (23 typically), plus a last line which is the status line. The screen window is a predefined number of lines on the screen that certain editing operations treat as a unit. Window size can be altered through the window size option, which is covered in the section "Setting Up Your vi Environment" in this chapter. The window can be as large as the screen less the status line.

Figure 2-1 shows an example of a window.

```
1  This is the first line, which is also known as home.
.
.
12  This is the middle line.
.
.
23 This is the last line.
24 This is the status line.
```

Figure 2-1 Excerpt of a Sample Window

Each number in the previous example represents the number of that line on the screen. The numbers do not actually appear on the screen, however.

Mark

A mark saves an exact location in a file. In command mode, you position the cursor at the desired location and use a command to save that position in a mark register. There is no visual mark in your file. To position the cursor at the mark at some later time, you can direct the cursor to move to the marked location by referencing the mark register. You can have as many as 26 locations marked in a file.

How vi Relates to the Document Formatter: DTK

Vi is neither a text formatting program nor a text processing program; it doesn't perform right justification, nor does it center text, provide a variety of fonts, or identify section heads. It was, however, designed for use with a document formatting program package called the Documenter's Tool Kit, DTK for short, which can be ordered as a separate software product.

There are some features in **vi** that are specifically relevant to DTK users. For example, in addition to the previously discussed set of **vi** text objects, there is also a section object that refers to the beginning of a section and the end of a section, both with respect to the current cursor position. A section is a unit of text that is labelled with a section header. The section text object is bounded by the section delimiter commands recognized by the DTK programs **nroff** and **troff**.

Furthermore, **nroff** and **troff** recognize a set of paragraph delimiter commands, which are the beginning and ending of a paragraph. DTK offers several varieties of document production oriented command sets that you can use to format and produce your documents. DTK supports the **mm** macro set. Refer to the DTK package of documentation (see the Preface for the titles) for more information on the DTK commands, and refer to the section "Setting Up Your vi Environment" later in this chapter for more information about recognized **mm** commands.

Setting Up vi Options

You can set editor options to customize your editing environment to suit your tastes. A subset of useful options is described here that may be beneficial for your work in **vi**. The examples of the editing operations shown in this chapter assume the setup of these options. A full list of all possible options is given in the section "Setting Up Your vi Environment," later in this chapter.

There are three methods for setting these options. Each method specifies how long the option will remain in effect.

- **EXINIT** environment variable.
- **.exrc** file.
- Last line mode.

The **EXINIT** environment variable remains in effect for a single log-in session (until you log out). The **.exrc** file is initialized each time you log in and for each subshell or new shell generated. The options set in last line mode remain in effect only for the current editing session (until you exit **vi**).

Setting the **EXINIT** environment variable is shown here. For information on the alternate methods, refer to the section "Setting Up your vi Environment," later in this chapter.

From the shell, you will assign the values contained within single quotation marks to the environment variable **EXINIT**. (More information on environment variables is given in *Using the DG/UX™ System*).

Bourne shell:

```
$ EXINIT='set wm=10 smd redraw report=1' ↵
$ export EXINIT ↵
```

C shell:

```
% setenv EXINIT 'set wm=10 smd redraw report=1' ↵
```

For the Bourne shell, you assign selected values to the **EXINIT** variable. A pair of single quotation marks (') surround the values. The variable is then exported to the environment with the **export** command. When making the assignment, make sure there are no spaces around the equal sign (=).

For the C shell, you set the same values to the variable, **EXINIT**. However, you use one command, **setenv**, to assign the values to the variable and export the variable to the environment.

To verify your settings, you can type the following command:

```
$ env ↵
EXINIT=set wm=10 smd redraw report=1
```

They will be in effect automatically only for the current log-in session.

Table 2-2 describes each of these options:

Table 2-2 Vi Editing Options Set for Examples in This Chapter

Option	Effect	Value
wrapmargin	Performs automatic new line at a defined column position from the right side of the screen.	**wm=10**
showmode	Displays message in status line indicating mode of operation.	**smd**
redraw	Redraws screen to show insertion of text and movement of existing text to the right.	**redraw**
report	Reports statistics in status line for all editing operations affecting one or more lines.	**report=1**

Useful Tips While Using vi

If you choose to try out some of these commands as you read about them in this chapter, you may make a few mistakes or you may decide to exit **vi** for a while. These four functions may be helpful to you during your **vi** work. They are:

- Undoing a command.
- Redrawing (or refreshing) your screen.
- Periodically saving the working buffer.
- Quitting **vi**.

These topics are discussed in detail in the following sections.

Undoing a Command (u)

If you type a command that produces a result you do not like or you prematurely press a key that causes an undesired command to execute, you can recover with these simple keystrokes:

From command mode, type the **u** command to undo the effect produced by the previous command.

Your original text will be restored to what it was before you issued the command. You can press this key repeatedly to restore and undo only the preceding command; this key acts like a toggle switch.

Refreshing Your Screen (Ctrl-L)

If your screen gets littered with spurious marks, which can result from modem "noise," or broadcast messages, you can refresh your screen in command mode by pressing Ctrl-L.

Vi will erase the marks from your screen and refresh the screen eliminating extraneous characters.

Periodically Saving the Working Buffer (:w)

You should perform periodic saves of the working buffer during a **vi** work session as a precautionary measure against system crashes. Here's what you can do:

- From command mode, press the colon (:) key to invoke last line mode. You will see this keystroke echoed on the status line of the screen.
- Type the **w** command and press the New Line key.

  Your working buffer will be saved in the filename specified when you invoked **vi** (if the filename already exists, its original contents are overwritten). The cursor will return to its previous position on the screen. Refer to "Commands to Write Files" for a list of variations on this command.

Quitting vi (ZZ, :wq, :q, :q!)

After you have tried some **vi** commands, you may decide to quit your session for a while. The easiest ways to quit **vi** and save the contents of the file are shown as follows:

From command mode, to save the current buffer and quit **vi**, type **ZZ**.

- In last line mode, you can also type the **:wq** command and press the New Line key.

- If you have made no changes to the current buffer, quit **vi** by typing the **:q** command and pressing the New Line key. If changes were made, **vi** issues a warning message that there was no write since the last change.
- Alternatively, in last line mode, you can quit the session without saving the buffer contents using **:q!** and pressing the New Line key.

A copy of your edited text is saved in a file with the name you assigned when you initialized **vi**. Your editing session with **vi** is terminated and you are free to use your terminal and the system in some other way.

A full list of methods for writing files is given in the section "Manipulating Files in Last Line Mode," later in this chapter.

Invoking vi

To invoke the editor **vi**, from the shell, you type the following command and supply a filename. **Vi** will create a file with the assigned name if it doesn't already exist. If a file with the assigned name already exists, **vi** will open it for editing. The filename you choose must follow standard DG/UX system file-naming conventions (refer to *Using the DG/UX™ System* for information on filenames).

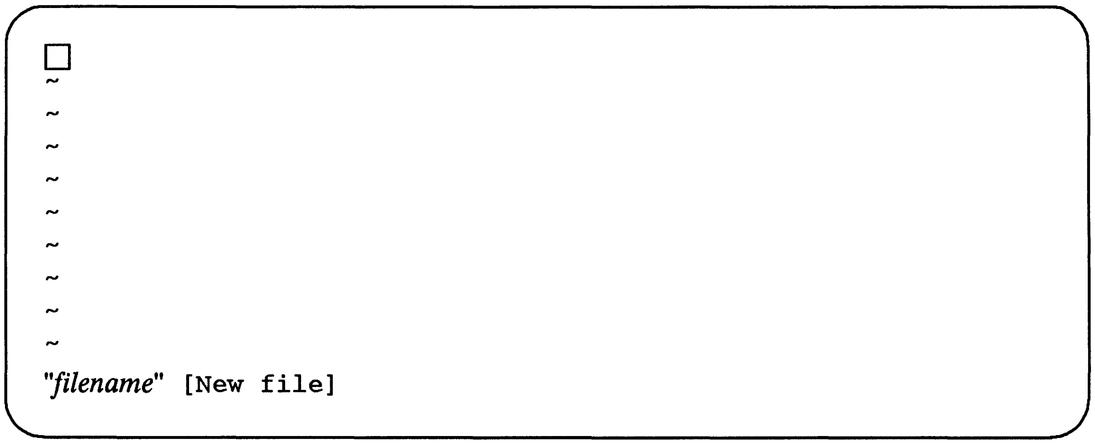
vi [*options*] *filename*

Table 2-3 defines the options.

Table 2-3 Vi Command-Line Options

Command	**Description**
view *argument*	You can substitute **view** for all **vi** commands to specify a read-only mode.
vi	Edits an empty buffer.
vi *filename*	Edits an existing file, starting at line 1, or creates one with *filename* if it doesn't exist.
vi + *filename*	Edits an existing file, starting at the last line.
vi *line-num filename*	Edits an existing file, starting at a specific line number.
vi +/*pattern filename*	Edits an existing file, starting at the first line containing the specified pattern (refer to Appendix A for information on regular expression pattern-matching).
vi *filename1 filename2 ...*	Edits several existing files in sequence. Starts with line 1 in the first file. When you finish editing a file, go to the next file by pressing the Escape key, **:n** (for next), and the New Line key
vi −r	Lists a summary of file(s) being edited that were automatically saved as a result of a system crash.
vi −r *filename*	Recovers a specific file that was in the working buffer prior to a system crash. As many as 12 lines or parts of lines may have been lost. Use this command from the directory containing the file being edited.
vi −t *label*	If you have a file named **tags** (which contains lines in the format *label file/string*), **vi** clears the working buffer to edit *file*, and positions the cursor on the line containing *string*. Refer to the section on producing and using a tags file in *Using the DG/UX™ System*.
vi −l *filename*	Turns on the lisp option (see the section on "Setting Up Your vi Environment") in the current **vi** work session.
vi −w*n filename*	Sets the screen window size to *n* lines, overriding the default which can be set using the window option (see the section titled "Setting Up Your vi Environment").

Figure 2-2 shows a typical screen that appears after you press the New Line key.



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

Figure 2-2 Typical Screen

The screen contains three important items:

Cursor (□)
The cursor symbol on the top line shows the point at which you can type input. Your cursor symbol may appear differently.

Tilde (~)
All other lines are marked with a tilde, the symbol designating lines past the end of the file. When you create an empty file, you will see the appearance of tildes on the left side of the screen.

Status line (*"filename"* `[New file]`)
Vi displays the status line on the bottom line of the screen — line 24 on most terminals. It contains information about the status of the working buffer, such as a message to alert you to the input mode status and error messages. From this line, you also issue commands in last line mode.

Vi Editing Operations

The rest of this chapter covers entering text in input mode and editing text in command mode. The major topics are:

- Entering text.
- Moving the cursor.
- Appending text.
- Inserting text.
- Opening a new line in text.
- Deleting text.
- Replacing text.
- Changing text.
- Moving text.
- Marking text.
- Searching for patterns.
- Manipulating files in last line mode.
- Setting up your **vi** environment.
- Writing and using macros.
- Performing other miscellaneous tasks.

Entering Text

Here's how you enter text. From command mode, you first need to give a command to signify that you want to enter text. In this case, you decide to "append" text with the **a** command after which you type your text. After you finish entering text, press the Escape key to signal and end of append mode. Remember that commands entered in command mode are not visible on the screen. An example follows:

Enter the **a** command for append mode.

Then, type this text:

> **Suddenly, we spotted whales in the distance. Daniel saw** ⤸
> **them first. "Hey look! Here come the whales!" he cried** ⤸
> **excitedly.**□

Then, press the Escape key to end entry mode.

If you make a mistake while entering text, the easiest way to erase a character you just typed is to press the Del key. Each time you press the Del key, the character that you back over is deleted. If the Del key does not operate, your terminal may use another character delete key (refer to *Using the DG/UX™ System* for information on selecting a line discipline). You may want to consult your system administrator for help.

Assuming that you set the automatic wrapmargin option, your text will break at the closest space-delimited word when it reaches the right margin. A new-line character is automatically appended to the end of each line when the cursor wraps. If you did not set this option, you will need to press the New Line key at the conclusion of each line. You should avoid ending a line with a space as some **vi** commands may produce unexpected results if they encounter a space at the end of a line.

Also, if the showmode option is set (refer to Table 2-35), the "`APPEND MODE`" message is displayed on the right side of the status line to let you know that **vi** is accepting your keystrokes as text input. When you press the Escape key to terminate input mode, this message disappears from the screen.

Moving the Cursor

With **vi** in command mode, you can position the cursor anywhere in the current working buffer. **Vi** accomplishes this by moving the cursor right and left on a line, up and down between lines, and backward and forward within the current window and the entire working buffer. As the cursor approaches the end of a line, it will normally wrap to the beginning of the next line. If you move the cursor backward to the beginning of the line, it will wrap to the end of the previous line.

You can also move the cursor through the working buffer by text object units — character, word, space-delimited word, line, sentence, and paragraph. You can also move through multiple text objects by preceding the cursor command with a number.

Each of these cursor movements is discussed:

- By a character.
- By a line.
- Within a line.
- By a word and space-delimited word.
- By a sentence.
- By a paragraph.
- Within the current screen window.
- From line to line (nondisplayed portion of file).

Vi must be in command mode before you can use the cursor commands. Once **vi** is in command mode, you can use any number of cursor commands to position the cursor in the working buffer (and your screen). When you type any of these commands, you will not see their literal appearance. Each keystroke will be interpreted as an instruction to move the cursor. You will see the effect it produces, such as moving the cursor right, one position at a time.

Moving the Cursor by a Character

Table 2-4 lists the basic character cursor movement commands.

Table 2-4 Commands to Move the Cursor by a Character

Command	Definition
l or →	Moves the cursor one character (or multiple characters) to the right.
h or ←	Moves the cursor one character (or multiple characters) to the left.

Each of these commands can be preceded by a number to indicate the number of characters you want to move the cursor. Examples of moving the cursor by the character are given in the following examples. Sample text follows:

$\texttt{Suddenly we spott}\boxed{\texttt{e}}\texttt{d the whales.}$

When you use the **l** command, the cursor moves one character to the right, as follows:

```
   Suddenly we spotte[d] the whales.
```

When you use the **5h** command, the cursor moves five characters to the left, as follows:

```
   Suddenly we s[p]otted the whales.
```

To accomplish the same character movements, you can use the space bar or rightarrow key (→) to move the cursor to the right. To move the cursor to the left, you can use the Ctrl-H or the leftarrow (←) key. You can also precede each command with a number to move the cursor multiple character positions.

The maximum number of characters you can specify is restricted by the line length. If the number of characters you specify (e.g., (**88→**)) exceeds the boundaries of the screen, the cursor goes to the end of the line. If the cursor is already at the end of the line, your terminal will beep and the cursor will remain stationary.

NOTE: On some terminal keyboards, you can press and hold the given cursor movement keys to produce a repeated cursor movement. If your terminal has a Repeat key, you may have to press and hold the cursor key with the Repeat key.

Moving the Cursor by a Line

You press the New Line key to move the cursor to the beginning of a new line and you use one of the following commands to move the cursor to the same column position in other lines. Table 2-5 lists the commands to move the cursor by the line:

Table 2-5 Commands to Move the Cursor by a Line

Command	Definition
j	Moves cursor down one line or multiple lines, staying in same column.
k	Moves cursor up one line or multiple lines, staying in same column.

Each of these commands can be preceded by a number to indicate the number of lines you want to move the cursor.

NOTE: The **j** command also maps to the down arrow (↓) key. For Data General terminals in DG mode and the C shell only, however, the ↓ key also maps to the Suspend key Ctrl-Z, which suspends temporarily the current **vi** session. If you use the *n*↓ command to move the cursor, you will suspend **vi** operation unless you have mapped the suspend function to some other key or have the **novice** option set (see a later section "Setting Up Your vi Environment" for more information about this option). Refer to *Using the DG/UX™ System* for information about line disciplines, and Chapter 1 in this manual for information on redefining process control keys.

Examples of moving the cursor by the line are given using the following sample text:

Suddenly, we spotted whales in
the dis$\boxed{t}$ance. Daniel was the first to see them.
"Hey look! Here come the whales!" he cried excitedly.

When you use the **j** command, the cursor moves one line down, with the cursor remaining in the same column position, as follows:

Suddenly, we spotted whales in
the distance. Daniel was the first to see them.
"Hey lo$\boxed{o}$k! Here come the whales!" he cried excitedly.

NOTE: The uparrow (↑) and downarrow (↓) keys can also be used to position the cursor up and down.

When you use the **2k** command, the cursor moves from its previous position up two lines as follows:

Suddenl$\boxed{y}$, we spotted whales in
the distance. Daniel was the first to see them.
"Hey look! Here come the whales!" he cried excitedly.

From the cursor position, (if there is no character in the same column of the line above or below), shown as follows,

Suddenly, we spotted whales in
the distance. Daniel was the first to see them.
"Hey look! Here come the whales!" he cried excite$\boxed{d}$ly.

when you use the **k** command, the cursor will move to the nearest column that does contain a character shown as follows:

Suddenly, we spotted whales in
the distance. Daniel was the first to see them$\boxed{.}$
"Hey look!" Here come the whales!" he cried excitedly.

You can also move the cursor to a line that is not currently displayed on the screen. If the specified line exists (the cursor is on line 25 and you ask for 24 lines above the current line **24k**), the screen will scroll until the desired line is within view. Alternatively, if you request a line that exceeds the size of the working buffer (you ask for **100j** and there are only 25 more lines beneath the current line), the terminal will beep and the cursor will remain stationary.

Moving the Cursor Within a Line

There are three methods to move the cursor within a line:

- To the beginning or end of the line.
- To a specific character in the line.
- To a specific column in the line.

Moving the Cursor to the End or Beginning of a Line

Table 2-6 lists the three commands to move the cursor to the beginning or end of the line:

Table 2-6 Commands to Move the Cursor to the Beginning or End of a Line

Command	Definition
$	Puts the cursor on the last character of a line.
0	Puts the cursor on the first character of a line.
^	Puts the cursor on the first nonblank character of a line.

The following examples show the movement of the cursor produced by each of these three commands.

To the End of a Line ($)

The following example shows how to move the cursor to the end of the line. The sample text follows:

$\texttt{Go to the }\boxed{\texttt{e}}\texttt{nd of the line!}$

When you use the $ command, the cursor goes to the end of the line as follows:

$\texttt{Go to the end of the line}\boxed{\texttt{!}}$

To the Beginning of a Line (0)

The following example shows how to move the cursor to the beginning of the line. The sample text follows:

```
Go to the begi$\boxed{n}$ning of the line!
```

When you use the **0** command, the cursor goes to the beginning of the line as follows:

```
$\boxed{G}$o to the beginning of the line!
```

To the First Nonblank Character of a Line (^)

In some cases the beginning of the line may contain one or more spaces. The **0** command would move the cursor to the beginning of the line to a blank position. To move the cursor to the first nonblank character, skipping over blanks, you would use the ^ command instead. The sample text follows:

```
Go to the first character
of the line
     that is $\boxed{n}$ot blank!
```

When you use the ^ command, the cursor goes to the first nonblank character as follows:

```
Go to the first character
of the line
     $\boxed{t}$hat is not blank!
```

Moving the Cursor to a Specific Character on a Line

Another way of positioning the cursor on a line is to search for a specific character on the current line. If you look for something that doesn't exist, your terminal will beep and the cursor will remain stationary. Character searches are case-sensitive unless you set the ignorecase option (refer to "Setting Up Your vi Environment" for more information).

Table 2-7 lists the ways to move the cursor to a specific character within a line.

Table 2-7 Commands to Move the Cursor to a Specific Character

Command	Definition
f*x*	Moves the cursor to the right to the specific character *x*.
F*x*	Moves the cursor to the left to the specific character *x*.
t*x*	Moves the cursor to the right, just to the left of the specific character *x*.
T*x*	Moves the cursor to the left, just to the right of the specific character *x*.
;	Repeats the previous operation in the same direction. The **;** command recalls the previous command and repeats it.
,	Repeats the previous operation in the opposite direction. The **,** command recalls the previous command and executes the complementary command (the one that searches in the opposite direction).

Each of these commands can be preceded by a number to indicate the number of characters you want to move the cursor.

The sample text follows:

$\boxed{\texttt{G}}$`o forward to the letter A on this line.`

When you use the **fA** command, **vi** searches to the right for the first occurrence of the letter "`A`" on the current line as follows:

`Go forward to the letter` $\boxed{\texttt{A}}$ `on this line.`

More sample text follows:

$\boxed{\texttt{G}}$`o forward to just left of the letter i on this line.`

When you use the **ti** command, the cursor moves to the left of the first occurrence of the letter "`i`" in the current line, as follows:

`Go forward to just left of the letter`$\boxed{\ }$`i on this line.`

With the cursor positioned as follows:

`Go backward to the letter k on` $\boxed{\texttt{t}}$`his line.`

when you use the **Fk** command, the cursor moves left to the previous occurrence of the letter "k" in the current line.

```
Go backward to the letter [k] on this line.
```

From the cursor position, when you use the **;** command, **vi** repeats the most recent character search command as follows:

```
Go bac[k]ward to the letter k on this line.
```

Moving the Cursor to a Specific Column on a Line (n|)

Columns are specified with an absolute number. This command is helpful when moving the cursor directly to a known column position in a table, form, or program. Column positioning is requested using this syntax:

n|

where:

n is a column number.

This command moves the cursor right or left to the specified column number *n*. Also, column specification is absolute. It is not relative to the cursor's current position.

In the following example, the cursor is in column 8:

```
Go forw[a]rd to column 20 in this line.
```

To move the cursor to absolute column 20, you would use the **20|** command to produce the following result:

```
Go forward to colum[n] 20 in this line.
```

This command moves the cursor forward and backward in a line. Keep in mind that your maximum cursor movement is restricted by the length of the line. If you request a column position that exceeds the line length, the cursor will go to the end of the line.

Moving the Cursor by a Word

Table 2-8 lists the commands that move the cursor by the word:

Table 2-8 Commands to Move the Cursor by a Word

Command	Definition
w	Moves the cursor forward to the first character in the next word. You may type **w** as many times as you want to reach the word you want.
W	Moves the cursor forward to the first character in the next space-delimited word.
b	If the cursor is positioned on the first character of a word, moves the cursor backward to the first character of the previous word. If the cursor is not positioned on the first character of a word, moves the cursor to the beginning of the current word. This command is the opposite of the **w** command.
B	This command is the same as the **b** command except that it applies to space-delimited words.
e	If the cursor is positioned on the end character of a word, moves the cursor forward to the end character of the next word. If the cursor is not positioned on the end character of a word, moves the cursor to the end character of the current word. This command is the counterpart to the **w** command.
E	This command is the same as the **e** command except that it applies to space-delimited words.

Each of these commands can be preceded by a number to indicate the number of words (or space-delimited words) you want to move the cursor. Sample text follow:

He as$\boxed{k}$ed, "What is the time?"

By using the **w** command, the cursor moves to the next word, which is a comma, as follows:

He asked$\boxed{,}$ "What is the time?"

By using the **3W** command, the cursor moves to the first character in the third space-delimited word, as follows:

He asked, "What is $\boxed{t}$he time?"

By using the **b** command, the cursor moves back to the first character in the previous word.

He asked, "What $\boxed{i}$s the time?"

By using the **3B** command, the cursor moves to the first character of the third space-delimited word in a backwards direction, as follows:

$\boxed{H}$e asked, "What is the time?"

By using the **e** command, the cursor moves to the final character in the next word, as follows:

H$\boxed{e}$ asked, "What is the time?"

By using the **3E** command, the cursor moves to the final character in the third space-delimited word, as follows: The word "He" is counted as one of three space-delimited words.

He asked, "What i$\boxed{s}$ the time?"

Moving the Cursor by a Sentence

Table 2-9 lists the commands to move the cursor by the sentence:

Table 2-9 Commands to Move the Cursor by a Sentence

Command	Definition
(	Moves the cursor to the beginning of the previous sentence. If the cursor is located within the paragraph, the cursor moves to the beginning of the current paragraph.
)	Moves the cursor to the beginning of the next sentence.

You can precede each of these commands with a number to indicate the number of sentences you want to move the cursor.

The following examples show moving the cursor a sentence at a time. Sample text follows:

Suddenly we spotted $\boxed{w}$hales in the
distance. Daniel was the first to see them.

When you use the (command, the cursor moves to the beginning of the previous sentence, as follows:

$\boxed{S}$uddenly we spotted whales in the
distance. Daniel was the first to see them.

When you use the) command, the cursor moves to the beginning of the next sentence, as follows:

```
Suddenly we spotted whales in the
distance.  [D]aniel was the first to see them.
```

Moving the Cursor by a Paragraph

Table 2-10 lists the commands to move the cursor by the paragraph:

Table 2-10 Commands to Move the Cursor by a Paragraph

Command	Definition
{	Moves the cursor to the beginning of the previous paragraph. If the cursor is located within a paragraph, the cursor will move to the beginning of the current paragraph.
}	Moves the cursor to the beginning of the next paragraph.

Each of these commands can be preceded by a number to indicate the number of paragraphs you want to move the cursor. The following examples show moving the cursor by the paragraph. Sample text follows:

```
Suddenly, we spotted whales in the
distance.  [D]aniel was the first to see them.

"Hey look!  Here come the whales!" he cried excitedly.
```

When you use the { command, the cursor moves to the beginning of the current paragraph, as follows:

```
□
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.

"Hey look!  Here come the whales!" he cried excitedly.
```

When you use the } command, the cursor moves to the beginning of the next paragraph.

```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.
□
"Hey look!  Here come the whales!" he cried excitedly.
```

Moving the Cursor Within the Current Window

The current window is another name for the visible portion of the working buffer (the part that you do not scroll up or down). Within the current window, you can move the cursor to the first non-blank character of the top, middle, or last lines. If one of these lines is blank, the cursor goes to the first position of the blank line. Table 2-11 specifies the commands that you can use to position the cursor on the top, middle, or last lines.

Table 2-11 Commands to Move the Cursor Within the Current Window

Command	Definition
H	Moves the cursor to the first nonblank character of the first line on the screen; moves the cursor to the first column if the first line is blank.
M	Moves the cursor to the first nonblank character of the middle line on the screen; moves the cursor to the first column if the middle line is blank.
L	Moves the cursor to the first nonblank character of the last line on the screen; moves the cursor to the first column if the last line is blank.

Examples of using these commands to position the cursor on the home, middle, and last lines assume a 23-line screen display. Sample text follows.

```
[S]uddenly, we spotted whales in the
distance.  Daniel was the first to see them.
.
.
.
"Hey look!  Here come the whales!" he cried excitedly.
Never had he witnessed such a sight.  They were swift, powerful,
.
.
.
and also graceful.  The herd moved as one.
```

By using the M command, the cursor moves to the first position of the middle of the current window, shown as follows.

```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.
.
.
.
"Hey look!  Here come the whales!" he cried excitedly.
Never had he witnessed such a sight.  They were swift, powerful,
.
.
.
and also graceful.  The herd moved as one.
```

By using the **L** command, the cursor moves to the beginning of the last line of the current window, shown as follows.

```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.
.
.
.
"Hey look!  Here come the whales!" he cried excitedly.
Never had he witnessed such a sight.  They were swift, powerful,
.
.
.
and also graceful.  The herd moved as one.
```

You can precede the **L** command with a number to indicate a line position before the last line in the current window such as **3L**.

By using the **H** command, the cursor moves to the home (row 1, column 1) position, shown as follows.

```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.
.
.
.
"Hey look!  Here come the whales!" he cried excitedly.
Never had he witnessed such a sight.  They were swift, powerful,
.
.
.
and also graceful.  The herd moved as one.
```

You can also precede the **H** command with a number to indicate some line position after the first line in the current window such as **3H**.

Moving the Cursor Outside the Current Window

This group of commands enables you to move the cursor to parts of the working buffer that are not in the current window. Four methods are available for locating lines.

- Requesting a relative line.
- Requesting a specific line.
- Scrolling.
- Paging.

Moving the Cursor to a Relative Line (+, -)

An alternative to moving the cursor by the line is to specify a relative number of lines in a positive (+) or negative (–) direction from the current cursor position. When the line is located, the cursor is put in the first nonblank character position in the line. Table 2-12 lists the commands that moves the cursor to a relative line:

Table 2-12 Commands to Move the Cursor to a Relative Line

Command	Definition
n+	The screen scrolls down in a positive (+) or forward direction by *n* lines. The cursor moves to the first nonblank position on the given line.
n–	The screen scrolls in a negative (–) or backward direction by *n* lines. The cursor moves to the first nonblank position on the given line.

The *n*+ command can also be expressed without using the plus (+) sign.

Examples follow:

13+	The cursor moves down 13 lines.
13 ↵	The cursor moves down 13 lines.
13–	The cursor moves up 13 lines.

For lines not located in the current window, the screen will shift up or down to reveal the wanted lines. If, however, you tried to move the window up to a nonexistent line (such as backwards 100 lines, expressed as 100-, in a 50-line file), the cursor would remain on the current line and the terminal would beep.

Moving the Cursor to a Specific Line

Table 2-13 lists the commands used for positioning the cursor on an absolute line number:

Table 2-13 Commands to Move the Cursor to a Specific Line

Command	Definition
*n*G	The cursor goes to line *n* where *n* represents the absolute line number you specified. If that line is not currently on the screen, the window will shift so that the desired line is displayed.
G	The cursor goes to final line in the working buffer and the window will shift so that the desired line is displayed on the screen.

You can find out the absolute line numbers by setting the number (**nu**) option from last line mode (this information is given in the section on "Setting Up Your vi Environment" later in this chapter). After you set the numbering option, each line in your file is preceded by an absolute line number. If, for example, you want to position the cursor on line 150 in your file, from command mode you would issue the **150G** command and the portion of the file containing the specified file is positioned within the current window. The cursor is positioned in the first column of line 150. To remove line numbers from your file, in last line mode, you can issue the **nonu** command.

Scrolling and Paging Text Through the Current Window

There are two ways you can display text that is not visible in the current window.

Scrolling	Rolls one-half a screen of text (11 lines by default) up or down in the buffer. Scrolling is best used when you want to read text continuously as one line at a time is sent to the screen.
Paging	Redraws the screen; shifting a complete page of text (23 lines by default) up or down in the buffer. Paging is faster than scrolling (depending on the baud rate and the screen size), but you do not see the context of the current screen as you do with scrolling.

The amount of text scrolled or paged is relative to the window size and the scroll size (for scrolling only). The window and scroll options can be set to customize window and scroll size (refer to the section "Setting Up Your vi Environment" later in this chapter).

Table 2-14 lists four commands that allow you to scroll and page the text in a file:

Table 2-14 Commands to Scroll and Page Text

Command	Definition
Ctrl-D	Scrolls down one-half a screen of text (by default), revealing text that is below the current window. If the current window is defined as 23 lines, then scroll size will be 11 by default.
Ctrl-U	Scrolls up one-half a screen of text (by default), revealing text that is above the current window. If the current window is defined as 23 lines, then scroll size will be 11 by default.
Ctrl-B	Pages up a full screen (23 lines by default), revealing text that is above the current window. Two lines are skipped between each screen for faster movement.
Ctrl-F	Pages down a full screen (23 lines by default), revealing text that is below the current window. Two lines from the previous screen are also displayed in the new screenful for continuity when reading forward.

When scrolling and paging, if there are not enough lines in the file to satisfy a scrolling or paging request, you will get what there is. If there are no more lines, your terminal will beep. If, for example, you're toward the bottom of a file and **vi** cannot scroll forward a full screen, the available lines will be moved to the current window, and tildes (~) will occupy the first columns of the empty lines to show the end-of-file. If the bottom of the file was already positioned in the current window and you tried to move a screen forward again, the screen would remain the same and the terminal would beep.

By default, a window contains 23 lines. The following sample text shows the current window with line numbers appearing in the left side of the screen.

NOTE: You can number your lines automatically by using the **number** editing option in last line mode (see the section "Setting Up Your vi Environment").

```
1  The kayaker pointed
2  the nose of his boat
.
.
.
22 downstream.  His name
23 was Mike.
```

By using the Ctrl-F command to page forward, you will cause the next page (22 lines) of undisplayed text to be presented in the current window. For continuity, however, there is a two-line overlap. Instead of seeing lines 23–45, you will see lines 22–44. The cursor can be positioned anywhere on the current screen.

```
22 downstream.  His name
23 was Mike.
.
.
.
43 Close call!
44 He paddled hard to avoid
```

Notice the two-line overlap.

By using the Ctrl-D command to scroll one-half screen up (moving in a backward direction), you would return one-half of the previous screen of text to the current window. One-half screen equals 11 lines.

```
9
10
11 He was preparing for
12 what he thought to be a simple rapid.
.
.
.
19 □
20
21
22 downstream.  His name
23 was Mike.
24 He knew no fear.
.
.
.
30 Quick recovery!
31 He resolved to be more careful.
```

Your next scroll or page actions will be relative to the current cursor position.

Moving the Cursor to a Marked Location

Using marks, you can flag significant areas in your file and save them in mark registers. After you mark a location with the mark command, you can then return the cursor to it automatically by specifying the register's name.

The mark command format follows:

m*register-name*

where:

m stands for "mark." You place the cursor at the position you want to mark, and then issue the **m** command.

register-name is the storage location to which you assign the marked position. You assign the location to a register that you name with a single letter, **a** through **z**.

The go-to-mark command format follows:

position-commandregister-name

where:

position-command can be either of the following:

‘ (backquote) goes to the exact cursor position saved.

’ (single quote) goes to the first nonblank position on the line containing the mark.

register-name is the marked location to which you assigned the marked position. You can use as many as 26 register names (**a** through **z**) per editing session.

The contents of the used registers are cleared when you quit **vi**.

Sample text follows.

```
Ski the summit
Ski 4 of the world's greatest ski areas on
1 lift package --- 8 different mountains ---
256 trails --- 53 lifts  ---  Located where
our airlines can take you!
What next$\boxed{?}$
```

By using the **ma** command, you can mark the current cursor position. This command marks the cursor's position at the question mark (?) and assigns the mark to register **a**.

Using the **’a** command from any file location, you can return the cursor to the first position in the marked line, as follows.

```
$\boxed{W}$hat next?
```

Using the **‘a** command from any file location, you can return the cursor to the exact marked location. This command (a single backquote) positions the cursor at the exact location marked.

```
What next$\boxed{?}$
```

NOTE: Storing a mark in a register with the **m** command overwrites that register's current contents.

Appending Text

The append commands put **vi** in input mode and place the text you type after the current character (the cursor position) or at the end of the current line. Table 2-15 lists two commands you can use to append text.

Table 2-15 Commands to Append Text

Command	Definition
a	Appends text after the cursor position.
A	Appends text at the end of the current line.

Before you can actually append any text, you must type one of the append commands, which puts **vi** in input mode. The key point to remember is that entry begins after the cursor.

Sample text follows:

```
Daniel saw the□whale spout.
```

Using the **a** command enables you to append text. In this sample text, you would type the **a** command immediately followed by the text "**white**" and then press the space bar to append the word "`white`" to "`the`". The "`APPEND MODE`" message will be displayed in the status line. After you finish appending text, you press the Escape key to return to command mode, and the message is removed from the status line.

The result appears as follows.

```
Daniel saw the white□whale spout.
```

If you had used the **A** command, the text would have been appended as follows:

```
Daniel saw the whale spout.white □
```

For either append command, with the cursor positioned at the end of a line, you can also press the New Line key once to start a new line of text, or multiple times to generate multiple blank lines.

Inserting Text

The insert commands put **vi** in input mode and place the text you type before the current character or at the beginning of the current line. Table 2-16 lists commands for inserting text.

Table 2-16 Commands to Insert Text

Command	Definition
i	Inserts text before the cursor.
I	Inserts text at the beginning of the current line.

Append and insert modes may appear the same but they are slightly different. The difference is that with insert mode, text entry begins before the cursor, not after. For example, assume the cursor is positioned as follows:

```
Daniel saw the whale spout.
```

Using the **i** command enables you to insert text. In this sample text, you would type the **i** command, press the space bar, and type the text **white** to insert the word "`white`" before "`whale`". The "`INSERT MODE`" message will be displayed in the status line. After you finish appending text, you press the Escape key to return to command mode, and the message is removed from the status line.

The result appears as follows.

```
Daniel saw the white whale spout.
```

If the **I** command had been used, the text would have been inserted as follows:

```
 whiteDaniel saw the whale spout.
```

NOTE: There is a space before the "w" in "`white`".

For both insert commands, with the cursor positioned at the end of the current line, you can also generate blank lines by pressing the New Line key any number of times. After you have inserted text, to return to the command mode of **vi**, press the Escape key.

When entering text using the insert or append commands, you may not know when to press the space bar to get the desired leading or trailing space for the new text. After you press the Escape key to end input mode, you may see that your spacing was off. If so, you can use the **u** command and try again or use the text deletion and insertion commands to correct the error. Since the **a** and **i** commands require that you use different spacing, you may want to get into the habit of using one or the other exclusively.

Opening a New Line for Text

Rather than adding text within a line of text, you may want to generate a completely new line between two existing lines. Table 2-17 lists two ways to open a blank line for text entry.

Table 2-17 Commands to Open a New Line

Command	Definition
o	Creates a blank line below the current line.
O	Creates a blank line above the current line.

The open commands create a blank line directly above or below the current line, and positions the cursor in the first column ready for text in input mode. Sample text follows:

```
Daniel saw the whale spout.
Daniel saw the white whale spout.
Daniel ate shark for dinner.
```

Using the **o** command enables you to open a line (below the line containing the cursor) for typing new text. In this sample text, you would type the **o** command followed immediately by the text **It looked like a geyser.** to open the new line and enter new text. The "OPEN MODE" message will be displayed in the status line. After you finish opening a line for new text, you press the Escape key to return to command mode, and the message is removed from the status line.

The result appears as follows.

```
Daniel saw the whale spout.
Daniel saw the white whale spout.
It looked like a geyser.
Daniel ate shark for dinner.
```

If you had used the **O** command instead of the **o** command, this result would be produced:

```
Daniel saw the whale spout.
It looked like a geyser.
Daniel saw the white whale spout.
Daniel ate shark for dinner.
```

The new text was entered on the line above (instead of below) the line containing the cursor.

Deleting Text

Table 2-18 lists the commands you can use to delete text.

Table 2-18 Commands to Delete Text

Command	Definition
x	Deletes a single character (or multiple characters) at the cursor position and then moves the cursor to the right; a shortcut for **dl**.
X	Deletes a single character (or multiple characters) starting at one position to the left of the cursor; a shortcut for **dh**.
dw	Deletes a single word (or multiple words) starting at the cursor position.
dW	Deletes a single space-delimited word (or multiple space-delimited words) starting at the cursor position.
d0	Deletes characters from (but not including) the cursor position to the beginning of the line.
D	Deletes all characters starting at the cursor position to the end of the current line.
d$	Another form of the **D** command; however, it takes a number for deleting multiple lines.
dd	Deletes an entire line (where the cursor is positioned) or multiple lines.
d(	Deletes from the cursor position (but not including the cursor position) to the beginning of a single sentence or multiple sentences.
d)	Deletes from the cursor position to the end of a single sentence (or multiple sentences).
d{	Deletes from the cursor position (but not including the cursor position) to the beginning of a single paragraph (or multiple paragraphs).
d}	Deletes from the cursor position to the end of a single paragraph (or multiple paragraphs).
d*posreg*	Deletes from the cursor position to the position of the mark saved in the named register.

For most of the delete commands, you can precede the command with a number to indicate the number of text objects to be deleted. Three delete commands that do not take numbers are: **d0** , **D**, and **d***posreg*. You can use a number for the first two commands, but it will have no effect.

Each delete command is demonstrated in the next sections.

Deleting Characters at the Cursor and to the Right

The **x** command deletes a single character at the cursor position and then moves the cursor to the right. Sample text follows.

```
The first annual research review will be at Hayes Hall.
```

By using the **x** command, you can delete one character marked by the cursor, shown as follows.

```
The first nnual research review will be at Hayes Hall.
```

More sample text follows.

```
The first annual research review will be at Hayes Hall.
```

By using the **7x** command, you can delete seven characters to the right of (and including) the cursor, shown as follows.

```
The first research review will be at Hayes Hall.
```

Deleting Characters to the Left of the Cursor

The **X** command deletes a single character starting at one position to the left of the cursor. Sample text follows.

```
The first research review will be at Hayes Hall.
```

By using the **7X** command, you can delete seven characters to the left of (and including) the cursor, shown as follows.

```
The research review will be at Hayes Hall.
```

Deleting Words

The **dw** command deletes a single word starting at the cursor position. Sample text follows.

```
The semi-annual research review will be at Hayes Hall.
```

By using the **5dw** command, you can delete five words to the right of (and including) the cursor, shown as follows.

```
review will be at Hayes Hall.
```

Notice that the word "`semi-annual`" is counted as three words.

Deleting Space-Delimited Words

The **dW** command deletes a single space-delimited word starting at the cursor position. Sample text follows.

```
The semi-annual research review will be at Hayes Hall.
```

By using the **5dW** command, you can delete five space-delimited words to the right of (and including) the cursor, shown as follows.

```
be at Hayes Hall.
```

Notice that the word "`semi-annual`" is counted as one space-delimited word.

Deleting from the Beginning of a Line

The **d0** command deletes characters from (but not including) the cursor position to the beginning of the line. Sample text follows:

```
The first semi-annual research review will be at Hayes Hall.
```

By using the **d0** command, you can delete all text between the beginning of a line to the current cursor position, shown as follows.

```
review will be at Hayes Hall.
```

Deleting to the End of a Line

The **D** command deletes all characters starting at the cursor position to the end of the line. Sample text follows.

```
The first semi-annual research review will be at Hayes Hall.
```

By using the **D** command, you can delete all text between the current cursor position to the end of the line, shown as follows.

```
The first semi-annual research review□
```

NOTE: Use of the **d$** command to delete to the end of the line performs the same function.

Deleting Entire Lines

The **dd** command deletes an entire line (where the cursor is positioned). Sample text follows.

```
One panelist will present her ideas on circuit design
concepts.
This panel discussion
will begin promptly at 2 pm.  Graduate students
are encouraged to attend.
```

By using the **3dd** command, you can delete three entire lines, starting from the current line (marked by the cursor), shown as follows.

```
will begin promptly at 2 pm.  Graduate students
are encouraged to attend.
```

You will see the message "`3 lines deleted`" in the status line assuming that the **report** option (last line mode) is set to 3 or less. See the section on "Setting Up Your vi Environment" later in this chapter for more information.

Deleting to the Beginning of a Sentence

The **d(** command deletes from the cursor position (but not including) the cursor position) to the beginning of a sentence. Sample text follows.

```
One panelist will present her ideas on circuit design
concepts.  This panel discussion
will begin promptly at 2 pm.  Graduate students
are encouraged to attend.
```

By using the **2d(** command, you can delete two sentences, starting from the current cursor position moving left to the beginning of the second sentence, shown as follows.

```
pm.  Graduate students
are encouraged to attend.
```

You will see the message "`4 lines deleted`" in the status line, assuming the **report** option (last line mode) is set to 4 or less. See the section on "Setting Up Your vi Environment" later in this chapter for more information.

Deleting to the End of a Sentence

The **d)** command deletes from the cursor position to the end of a sentence. Sample text follows.

```
One panelist will present her ideas on circuit design
concepts.
This panel discussion
will begin promptly at 2 pm.  Graduate students
are encouraged to attend.
```

By using the **2d)** command, you can delete two sentences, starting from the cursor position moving right through the end of the second sentence, shown as follows.

```
One panelist will present her ideas on circuit design
concepts.
This panel discussion
will begin promptly at 2□
```

You will see the message "`3 more lines`" in the status line assuming the **report** (last line mode) option is set to 3 or less. See the section on "Setting Up Your vi Environment" later in this chapter for more information.

Deleting to the Beginning of a Paragraph

The **d{** command deletes from the cursor position (but not including the cursor position) to the beginning of a paragraph. Sample text follows.

```
One panelist will present her ideas on circuit
design concepts.  This panel discussion will begin promptly
at 2 pm.  Graduate students are encouraged to attend.

A workshop will be in the Commons Room afterwards.

She will also tour the circuit design lab.
```

By using the **2d{** command, you can delete from the cursor position moving left through the beginning of the second paragraph, shown as follows.

```
workshop will be in the Commons Room afterwards.

She will also tour the circuit design lab.
```

You will see the message, "`6 lines deleted,`" in the status line assuming the **report** option (last line mode) is set to 6 or less. See the section on "Setting Up Your vi Environment" later in this chapter for more information.

Deleting to the End of a Paragraph

The **d}** command deletes from the cursor position to the end of a paragraph. Sample text follows.

```
One panelist will present her ideas on circuit
design concepts.  The panel discussion will begin promptly
at 2 pm.  Graduate students are encouraged to attend.

A [w]orkshop will be in the Commons Room afterwards.

She will also tour the circuit design lab.
```

By using the **2d}** command, you can delete from the cursor position moving right through the end of two paragraphs, shown as follows.

```
One panelist will present her ideas on circuit
design concepts.  The panel discussion will begin promptly
at 2 pm.  Graduate students are encouraged to attend.

[A]
```

You will see the message "`3 lines deleted`" in the status line assuming the **report** option (last line mode) is set to 4 or less. See the section on "Setting Up Your vi Environment" later in this chapter for more information.

Marking Text for Deletion

With the mark command, you can delete text that extends from a specific mark (whose location is saved in a register) to the cursor position.

You put the cursor at the location you want to mark and save that location in a mark register using this command format:

m*register-name*

where:

m represents "mark."

register-name identifies the register in which the marked location is saved. Valid register names are **a** through **z**.

You can delete text from the mark to the cursor position using a delete command in the following form.

d*positionregister*

where:

d represents the delete command; it removes text from the current line to the mark saved in the named register.

position refers to either of two symbols used to mark the end boundary for deletion.

> **‘** (backquote) marks the exact cursor position.
>
> **’** (single quote) marks the line the cursor is on.

register identifies the register containing the appropriate mark. Valid register names are **a** through **z** and **A** through **Z**.

Sample text follows.

```
One panelist will present her ideas on circuit
design concepts.  The panel discussion will begin promptly
at 2 pm.  Graduate students are encouraged to attend.

A workshop will be in the Commons Room afterwards.

She will also tour the circuit design lab.
```

By using the **mz** command, you can mark the cursor position and store the position in register **z**.

Let’s say you wanted to delete text from the mark to the first line (beginning with "`One`".) You would position the cursor on the "`O`" in "`One`", shown as follows.

```
One panelist will present her ideas on circuit
design concepts.  The panel discussion will begin promptly
at 2 pm.  Graduate students are encouraged to attend.

A workshop will be in the Commons Room afterwards.

She will also tour the circuit design lab.
```

Using the **d‘z** command, you can delete from the cursor position to the mark.

The result appears as follows.

```
A□

She will also tour the circuit design lab.
```

Using the **d'z** command, you can delete from the mark through the entire marked line, shown as follows.

```
□
She will also tour the circuit design lab.
```

Recovering Deleted Lines

Sometimes, through a hasty action or an inadvertent mistake, you can delete lines that can't be recovered with the **u** or **U** commands. You could have committed this mistake about five commands ago, for example. **Vi** offers an error recovery mechanism to retrieve a deletion if it occurred within the last nine editing commands. **Vi** has nine number registers used for storing the last nine text deletions performed with a delete command (and also the delete-and-put command). Refer to the section on "Moving Text" later in this chapter for more information on the delete-and-put command. The delete-text registers are numbered from **1** through **9**. The line-recover command takes this form:

"*nput-command*

where:

the double quotation mark (") symbolizes a named register.

n is a register number (**1–9**).

put-command is either **p** (put after the cursor) or **P** (put before the cursor).

With the cursor on the position at which you want the lost lines restored, you would type the command. If you can't remember the correct value for *n*, you might start with 1 and work your way back to 9. An example of the quick way to do this is:

```
"1p
u.
u.
u.
u.
```

You are starting at the register 1. If it retrieves unwanted text, you can type an undo command (**u**) followed by a dot command (.). The dot will increment the register by 1 and repeat the command. You can type (or repeat) up to eight **u.** commands as needed to get to the desired text. You can retrieve no more than the nine most recent delete operations.

Replacing Text

The replace commands cause the new text you type to overwrite (or replace) existing text. Table 2-19 lists two commands to replace text.

Table 2-19 Commands to Replace Text

Command	Definition
*n***r**	Causes the next character you type to overwrite the character at the cursor position (or multiple characters beginning at the cursor). After you type that character, **vi** automatically returns to command mode. You do not have to press the Escape key.
R	Starting at the cursor position, overwrites existing text until you press the Escape key to return **vi** to command mode. If the end of the line is reached, this command will append the additional input as new text.

You can precede the **r** command with a number to indicate the number of characters to be overwritten.

Replacing a Character

The **r** command cause the next character you type to overwrite the character at the cursor position. Sample text follows.

`There are more than 75 ou`$\boxed{\texttt{e}}$`door Olympic events.`

Using the **r** command enables you to replace text. In this sample text, you would type the **r** command followed immediately by the letter **t** to replace the character "e" with "`t`". The "`REPLACE 1 CHAR`" message is displayed in the status line when you use the **r** command. It is removed when you type the replacement character.

The result appears as follows.

`There are more than 75 ou`$\boxed{\texttt{t}}$`door Olympic events.`

The cursor is stationary, and command mode remains in effect.

In addition to replacing a single character, you can also replace *n* adjacent characters with *n* instances of the same character.

Sample text follows.

`Spectators numbering `$\boxed{\texttt{1}}$`00000 are expected to attend.`

Using the **6r9** command, you can replace "`100000`" with "`999999`" (or replace the six numbers "100000" with six instances of the digit "9"). The "`REPLACE WITH 1 CHAR`" message is displayed in the status line when you use the **r** command, and it is removed when you type the replacement character.

The result appears as follows.

```
Spectators numbering 999999 are expected to attend.
```

Command mode is still in effect.

Replacing Multiple Characters

The **R** command overwrites existing text starting at the cursor position and ending when you press the Escape key to return to command mode. Sample text follows.

```
Spectators numbering 999999 are expected to attend.
```

Using the **R** command enables you to replace multiple characters. In this sample text, you would type the **R** command followed immediately by the text **rose to their feet to applaud** to replace "`numbering 999999 are expected`". The **R** command replaces multiple characters until you press the Escape key to terminate entry mode, returning **vi** to command mode. The message "`REPLACE MODE`" is displayed in the status line after you type **R** and while you type the replacement text. After you have completed entering text, you press the Escape key to return to command mode, and the message is removed from the status line.

The result appears as follows.

```
Spectators rose to their feet to applaud to attend.
```

Notice that the old text "`to attend`" remains on the screen. To get rid of unwanted text, you should use a delete command. (See the section on "Deleting Text" in this chapter for more information.)

Remember that you are overwriting on the current line only. If your new text exceeds the boundaries of the old text from the current line, the excess text entered is appended. This means that you will not overwrite existing text. The following example illustrates the use of append following overwriting. Sample text follows.

```
Spectators □numbering 999999 are expected to attend.

The Olympic Committee began work on the festival years ago.
```

In this sample text, you would type the **R** command followed immediately by the text **jammed into the arena for the gymnastics event.** to replace the existing text and append to the end of text.

The result appears as follows.

```
Spectators jammed into the arena for the gymnastics event, which
featured the defending champion.

The Olympic Committee began work on the festival years ago.
```

Text on the current line was completely overwritten; however, the text exceeding the current line was appended. Notice that the subsequent text was not overwritten.

Changing Text

The change commands replace existing text with new text. The new text does not have to occupy the same amount of space as the old text. The change command will mark the specified amount of text to be changed by placing a dollar sign ($) on the final character in the range and put **vi** in input mode. After you finish entering the new text, you can press the Escape key to return **vi** to command mode. If the changed text exceeds the text area marked for change, additional text will be inserted. If the changed text occupies less space than the text area marked for change, when you press the Escape key, the old text is deleted and the excess space is squeezed.

Table 2-20 lists the change commands:

Table 2-20 Commands to Change Text

Command	Definition
s	Substitutes the current character (or multiple characters, starting with the current character) for another character(s). Input mode is terminated by pressing the Escape key.
cl	Same as *n***s**.
cw	Changes a single word (including a fragment) or multiple words to new text.
cW	Changes a single space-delimited word (including a fragment) or multiple blank- delimited words to new text.
cc	Changes all characters in the current line (or multiple lines) beginning with the current line to new text.
S	Same as *n***cc**.
C	Changes the remaining characters in the current line from the cursor to the end of the line (or multiple lines) to new text.
c$	Same as *n***C**.
c0	Changes the characters in the current line from the cursor to the beginning of the line (or multiple lines) to new text.
c(	Changes from the cursor position (but not including the cursor) to the beginning of a single sentence (or multiple sentences) to new text.
c)	Changes from the cursor position to the end of a single sentence (or multiple sentences) to new text.
c{	Changes from the cursor position (but not including the cursor) to the beginning of a single paragraph (or multiple paragraphs) to new text.
c}	Changes from the cursor position to the end of a single paragraph (or multiple paragraphs) to new text.
c*posreg*	Changes from the cursor position to the mark in a forward direction.

For all of the change commands, except the **c***posreg* command, you can precede the command with a number to indicate the number of text objects to be changed.

Substituting Characters

The **s** command substitutes the current character for another character until you press the Escape key to return to command mode. Sample text follows.

```
She gripped the branch with force.
```

Using the **s** command enables you to substitute one character for another and continue inserting text. After you type the **s** command, the "CHANGE MODE" message is displayed on the status line, and a dollar sign (**$**) marks the beginning of text to be substituted, shown as follows.

```
She $ripped the branch with force.
```

Then you can type the substituted characters, such as **st**. After you finish substituting text, you press the Escape key to return to command mode.

The result appears as follows.

```
She stripped the branch with force.
```

The appearance of the substitute and insert action is terminal dependent. On some terminals, if you do not set the **redraw** option (last line mode), when you type "**st**", the cursor may appear to overwrite the existing "r". (See the section on "Setting Up Your vi Environment" later in this chapter for more information.) After you press the Escape key, you will see the reappearance of the existing character.

An example of specifying a number with the **s** command is demonstrated with the following sample text.

```
Karen took a deep breath and skated across the ice.
```

In this sample text, you would type the **4s** command followed immediately by the text **tipto** to substitute the four characters "tipt" and insert one character "o". After you finish supplying the substitution text, you press the Escape key to return to command mode.

The result appears as follows.

```
Karen took a deep breath and tiptoed across the ice.
```

Changing Words

The **cw** command changes the current word, allowing you to continue inserting text until you press the Escape key to return to command mode. Sample text follows.

```
The kayaker peeled out and washed down the rapids.
```

In the sample text, you would type the **cw** command followed immediately by the text **catapulted** to change "washed" to "catapulted". After you finish the changed text, you press the Escape key to return to command mode.

The result appears as follows.

```
The kayaker peeled out and catapulted down the rapids.
```

Notice that the length of the changed text does not have to match the length of the old text. After you pressed the Escape key, text to the right is automatically justified.

Sample text for changing three words follows.

```
The kayaker peeled out and washed down the rapids.
```

In this sample text, you would type the **3cw** command followed by the text **a narrow chute**, to change the existing three words to three new words. After you finish changing text, you press the Escape key to return to command mode.

The result appears as follows.

```
The kayaker peeled out and washed down a narrow chute.
```

Notice that the period counted as a word.

Changing Space-Delimited Words

The **cW** command is identical to the **cw** except that the text object changed is a space-delimited word (adjacent punctuation is considered part of a word). Sample text follows.

```
The kayaker peeled out and washed down the rapids.
```

In the sample text, you would type the **3cW** command followed immediately by the text **through a boulder garden** to change three existing space-delimited words to new text. After you finish changing text, you press the Escape key to return to command mode.

The result appears as follows.

```
The kayaker peeled out and washed through a boulder garden.
```

Changing all Characters in the Current Line

The **cc** command allows you to change the current line with the cursor positioned anywhere on the line. If your new text exceeds the text to be changed, you can continue entering text until you press the Escape key to end input mode. Sample text follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
```

From the cursor position, you would type the **cc** command followed immediately by the following text:

If you care to put in some extra ↵
hours for extra pay and if you're in the vicinity

The **cc** command erases the current line, and the old text is changed to the new text.

After you finish the changed text, you press the Escape key to return to command mode.

The result appears as follows.

```
Dispatch to All Cab Drivers:

If you care to put in some extra
hours for extra pay and if you're in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
```

Changing to the End of a Line

The **C** command changes text from the cursor position to the end of the line. This command places a dollar sign ($) over the final character to be changed. If your new text exceeds the old text, it will be appended in input mode until you press the Escape key. Sample text follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
```

In this sample text, you would type the **C** command followed immediately by this new text.

want overtime work and will be in the area

The **C** command erases the current line, and the old text is changed to the new text.

After you finish the changed text, you press the Escape key to return to command mode.

The result appears as follows.

```
Dispatch to All Cab Drivers:

If you want overtime work and will be in the area
of the Central City Coliseum tonight,
you are in for big business.
```

Changing to the Beginning of a Sentence

The **c(** command changes text from the cursor position to the beginning of a single sentence or multiple sentences in a left direction. Sample text follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.
```

In this sample text, you would type the **2c(** command followed immediately by this text:

If you care to put in some extra ↵
hours for extra pay and if you're ↵
in the vicinity of the

Space once following the text.

The **c(** command erases the current sentence, and the text you type is inserted before the cursor, pushing existing text on the current line to the next line.

After you finish the changed text, you press the Escape key to return to command mode.

The result appears as follows.

```
Dispatch to All Cab Drivers:

If you care to put in some extra
hours for extra pay and if you're
in the vicinity of the□City Coliseum tonight,
you are in for big business.
There is a record sellout.
```

Changing to the End of a Sentence

The **c)** command is the complement of the **c(** command; it changes text from the cursor to the end of a single sentence or multiple sentences in a right direction. Sample text follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.
```

In this sample text, you would type the **c)** command followed immediately by this new text:

Galaxy Superdome tonight during ↵
the graveyard shift, do we have a job ↵
for you. It's a concert!

The **c)** command erases from the cursor to the end of a sentence, and the old text is changed to the new text.

After you finish the changed text, you press the Escape key to return to command mode.

On some terminals an at (@) sign may appear in the left margin signifying a deleted line. The cursor marks the text entry point.

The result appears as follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Galaxy Superdome tonight during
the graveyard shift, do we have a job
for you.  It's a concert!
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.
```

Changing to the Beginning of a Paragraph

The **c{** command changes text from the cursor to the beginning of a single paragraph or multiple paragraphs in a left direction. Sample text follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.
```

In this sample text, you would type the **c{** command followed immediately by this new text:

Taxi Drivers on the west side: ↵
A big concert is playing at ↵
the

Space once after the text.

The **2c{** command erases the text from the cursor the beginning of two paragraphs, and the old text is changed to the new text.

After you finish the changed text, you press the Escape key to return to command mode.

The result appears as follows.

```
Taxi Drivers on the west side:
A big concert is playing at
the Central City Coliseum tonight,
you are in for big business.
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.
```

Changing from the Cursor to the End of a Paragraph

The **c}** command changes text from the cursor position to the end of a paragraph or multiple paragraphs in a right direction. Sample text follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.

I can hear those meters ticking now.

Your friendly dispatcher, Rose
```

In this sample text, you would type the following **2c}** command followed immediately by this new text:

Grande Hotel, you might want ↵
to park your cars close by. There's ↵
a reception for the president of ↵
Galax Corp. Cabs will be needed ↵
to drive guests across town.

The **2c}** command erases text from the cursor position to the end of two paragraphs, and the old text is changed to the new text.

After you finish the changed text, you press the Escape key to return to command mode.

The result appears as follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Grande Hotel, you might want
to park your cars close by.  There's
a reception for the president of
Galax Corp.  Cabs will be needed
to drive guests across town.

Your friendly dispatcher, Rose
```

Changing Marked Text

The change mark command changes text from the cursor position to the mark in a forward direction. Sample text follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Grande Hotel, you might want
to park your cars close by.  There's
a reception for the president of
Galax Corp.  Cabs will be needed
to drive guests across town[.]

Your friendly dispatcher, Rose
```

You must first mark a location in your file. In this sample text, you would mark the cursor position with **ma** command, which assigns the cursor position to a mark register named **a**. You can use register names **a** through **z** to save marked positions. Defined marks are valid for the current **vi** session (until you log out).

Then, position the cursor in another file location to mark the boundary of the text to be changed. In this example, you can move the cursor to the "T" in "There's", shown as follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Grande Hotel, you might want
to park your cars close by.  [T]here's
a reception for the president of
Galax Corp.  Cabs will be needed
to drive guests across town.

Your friendly dispatcher, Rose
```

In this sample text, you would then type the **c'a** command followed immediately by this new text:

That area has recently been ↵
approved as a cab zone.

After you finish the changed text, you press the Escape key to return to command mode.

The result appears as follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Grande Hotel, you might want
to park your cars close by.
That area has recently been
approved as a cab zone[.]

Your friendly dispatcher, Rose
```

Moving Text

There are two types of commands for moving text from one area in the working buffer to another area; they are:

delete-and-put	Removes text from one location in the working buffer and puts it in another location in the working buffer.
yank-and-put	Makes a copy of text from one location in the working buffer and puts it in another location in the working buffer.

Vi uses several different registers for storing deleted and yanked text, which is covered in the next section.

The general procedure for the delete-and-put and the yank-and-put operations is the same:

- Identify the text being deleted or yanked using a delete or yank command.
- Position the cursor at the location for the moved or copied text.
- Issue the appropriate put command to position the text in the desired location.

The specific commands used for deleting, yanking, and putting text are given in this section.

Using Registers for Storing Text

There are three types of registers for storing deleted or yanked text that you can put in another location in the working buffer; they are:

general register	Saves the most recently deleted or yanked text implicitly.
numeric register	Saves the nine most recently deleted blocks of text implicitly.
alphabetic register	Saves deleted or yanked text explicitly for repeated use.

By default, all yanked or deleted text is saved automatically in a general register. Each time you perform a delete or yank operation, the deleted or yanked text will overwrite the previous contents of the general register. Also, the text you delete is saved implicitly in a number register. You do not have to assign a number register name; **vi** does it automatically.

With alphabetic registers, you can save text from delete and yank operations to which you assign a single character name so that you can use that text repeatedly. Table 2-21 lists the two types of named registers (number and alphabetic).

Table 2-21 Named Registers for Deleted and Yanked Text

Register Name	What It Does
a – z	Saves text in specific register, overwriting previously saved text in that register. To append text to an existing register, you simply use the register name's uppercase equivalent. For example, if you wanted to append more text to existing text register **a**, you would use register **A**. You can append as much text as you want to an existing register. Note that each new group of appended text is placed on a new line in the register; you cannot append text to an existing line in a register. Legal register names for appended text are **A–Z**.
1 – 9	Stores deleted text implicitly. You will not assign explicitly deleted text to these registers; it is stored there automatically whenever you use a delete command. However, you can recover deleted text and put it in another location in the working buffer by using its register. Since there are only nine numeric registers, only the last nine deletions can be stored. Therefore, you cannot recall text from the tenth or more delete operations back (unless you placed the text in an alphabetic register). The most recent deletion will be saved in register **1**; the oldest deletion will be saved in register **9**.

With text saved in specific registers, you can recall and put the text anywhere in the working buffer and at any time during a **vi** work session. The contents of all registers remain only until you quit **vi** with the **ZZ** or **:wq** commands.

The Delete Commands

The delete commands described in the section titled "Deleting Text" are used to erase text. Deleted text is stored automatically in the general register and a number register for subsequent put operations. Alternatively, you can also save deleted text in an alphabetic register for subsequent repeated put operations.

The command format for the delete operation using the general register follows:

delete-command

The command format for the delete operation using an alphabetic register follows.

n"register-namedelete-command

where:

n represents the number of text objects to be affected.

The quotation mark (") signifies a named register.

Legal alphabetic registers are **a** through **z**; and to append to an existing register, **A** through **Z**.

The delete commands are covered in the section "Deleting Text" in this chapter.

The Yank Commands

The yank commands are identical to the delete commands, except that text is not deleted from the working buffer. Rather, a copy is placed in the general register or a named register for subsequent puts. Table 2-22 lists the yank commands.

Table 2-22 Commands to Yank Text

Command	Definition
Y	Saves a copy of the current line (or multiple lines).
yy	Same as Y.
y_	Same as Y.
yw	Saves a copy of a single word (or fragment) or multiple words.
yW	Saves a copy of a blank-delimited word (or multiple blank-delimited words).
y0	Saves a copy of text from the left of the cursor (not including the cursor position) to the beginning of the line.
y$	Saves a copy of text from the cursor position to the end of the line (or multiple lines).
y(	Saves a copy of the text from the cursor to the beginning of a single sentence (or multiple sentences).
y)	Saves a copy of the text from the cursor (but not including the cursor) to the end of a single sentence (or multiple sentences).
y{	Saves a copy of the text from the cursor to the beginning of a single paragraph (or multiple paragraphs).
y}	Saves a copy of the text from the cursor to the end of a single paragraph (or multiple paragraphs).
m*reg*	Marks a position on a line.
y*posreg*	Saves a copy of the text from the marked position (either the entire line or the exact position saved in the register) to the cursor position.

All yank commands can be preceded by a number to indicate the number of text objects to be affected except the **y0** and y*posreg* commands. If yanked text is being saved in an alphabetic register, then that register name must precede the yank command.

The command format for the yank operation using the general register follows:

yank-command

The command format for the yank operation using an alphabetic register follows:

n"register-nameyank-command

where:

n represents the number of text objects to be affected.

The quotation mark (") signifies an alphabetic register.

Legal alphabetic register names are **a** through **z**; for appended text, **A** through **Z**.

The yank commands are covered in the preceding table in this section.

The Put Commands

The put commands copy the text from either the general, number, or alphabetic registers into the desired location in the working buffer. Text saved in the number and the alphabetic registers can be put in multiple locations in the working buffer at any time during a **vi** work session.

Each time you perform a yank or delete operation using the general register, that text will overwrite the previous contents. Therefore, it's a good idea to limit the operations performed between a yank (or delete) and put operation to cursor positioning commands. Any interfering commands will alter the contents of the general register and may give you unexpected results when you use the put command.

However, if you are putting text saved in a alphabetic register, there are no restrictions on any intervening commands you use between the delete or yank operations and the put operation.

The general format for the put command using text saved in the general register follows.

put-command

The general format for the put command using text saved in an alphabetic register follows.

"register-nameput-command

where:

The quotation mark (") signifies a named register.

The *register-name* is the name that corresponds to the text that was deleted or yanked. Register names are not case-sensitive when you are putting text. However, you must be careful with case sensitivity when you yank or delete to a named register. Table 2-23 lists the two forms of the put command:

Table 2-23 Commands to Put Text

Command	Definition
p	For characters and words, puts the saved text after the current character or word. For lines, sentences, and paragraphs, puts the saved text on the line after the current line (containing the cursor).
P	For characters and words, puts the saved text before the current character or word. For lines, sentences, and paragraphs, puts the saved text on the line before the current line (containing the cursor).

Examples of the Delete-and-Put and the Yank-and-Put Operations

Table 2-24 gives examples of the commands used for the delete-and-put and yank-and-put operations using general, number, and alphabetic registers.

Table 2-24 Examples of Delete-and-Put and Yank-and-Put Operations 30

Command	Description
5dw	Deletes five words and saves them in the general register and a number register implicitly. The most recently deleted text is stored in number register **1**.
p	Puts the saved text from the general register on the current line, following the cursor position.
P	Puts the saved text from the general register on the current line, before the cursor position.
2d{	Deletes two paragraphs from the cursor position to the beginning of the second paragraph from the cursor. This deleted text is saved in the general register and register **1** automatically; the previous deletion is now in register **2**.
"2P	Retrieves the deleted text from register **2** and puts it on the current line, before the cursor position.
5"zy}	Yanks five paragraphs from the cursor to the end of the fifth paragraph and saves them in alphabetic register **z**.
2"Zyw	Yanks two words from the cursor position and appends them to existing register **Z**.
"ZP	Puts the text saved from register **Z** on the line above the cursor position.

Two examples using registers are explained fully.

Deleting and Putting Text

The **dw** command deletes text from the cursor position to the end of a word. Sample text follows.

```
The semi-annual research review will be at Hayes Hall.
This conference is being planned now.
```

By using the **5dw** command, you can delete five words, starting at the cursor and moving right.

The result appears as follows.

```
review will be at Hayes Hall.
This conference is being planned now.
```

Notice that "`semi-annual`" is considered three words.

Saving Deleted Text in the General and Number Registers

Deleted text from the preceding example is saved automatically in the general register and in a number register. In the previous example, you deleted five words using the **5dw** command. Assuming that this deletion is the only one performed in this **vi** session, the deleted text is stored in register **1**. The deleted text follows.

```
The semi-annual research
```

Putting the Saved Text After the Cursor

If you want to insert this deleted text after the current cursor position in another buffer location, you can position the cursor there, and retrieve the contents of the general register or the numeric register.

Let's say you positioned the cursor on the "`T`" in "`This`", shown as follows.

```
review will be at Hayes Hall.
This conference is being planned now.
```

Since the previous delete operation is the only one in question, you can use the **p** command (which retrieves the contents of the general register), or you can specify the appropriate number register explicitly, in this example **"1p**.

This result is produced.

```
review will be at Hayes Hall.
TThe semi-annual research□his conference is being planned now.
```

The cursor is positioned on the final character of the inserted text. **Vi** remains in command mode.

Putting the Saved Text Before the Cursor

If you want to insert the saved text before the cursor, you position the cursor at the location where you want to put the saved text and type either **P** or **"1P**.

Let's say you positioned the cursor on the "T" in "This", shown as follows.

```
review will be at Hayes Hall.
This conference is being planned now.
```

Since the previous delete operation is the only one in question, you can use the **P** command (which retrieves the contents of the general register), or you can specify the appropriate number register explicitly, in this example **"1P**.

This result is produced.

```
review will be at Hayes Hall.
The semi-annual research□This conference is being planned now.
```

The cursor is positioned on the final character of the inserted text. **Vi** remains in command mode.

Yanking and Putting from an Alphabetic Register

In the following example, you want to yank two paragraphs extending from the cursor position to the end of the second paragraph and store it in an alphabetic register named **a**. Sample text follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.

I can hear those meters ticking now.

Your friendly dispatcher, Rose
```

By using the **2"ay}** command, you will yank (**y**) two (**2**) paragraphs extending from the cursor position to the end of the second paragraph (**}**) and store the text in a register (**"**) named **a**.

After you issue this command, the cursor will remain stationary and the message, "`8 lines yanked`" will be displayed on the status line:

The Yanked Text

The following is the text that is stored in register **a** with the command given in the preceding example:

```
If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.

I can hear those meters ticking now.
```

Destination for Yanked Text

After you have saved the text to be moved, you then position the cursor at the location in the buffer where you want to insert the text. In this example, you want to move the yanked text to the line following "`Your friendly dispatcher, Rose`" shown as follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.

I can hear those meters ticking now.

Your friendly dispatcher, Rose
```

Putting the Text in the Destination

After you position the cursor at the place you want to put the saved text, you can issue this command to actually perform the move operation: **"ap**.

where:

" is the symbol for a register.

a is the name of the alphabetic register in which the saved text is located.

p is the put command that positions the text on the line following the cursor position.

The final result follows.

```
Dispatch to All Cab Drivers:

If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.

I can hear those meters ticking now.

Your friendly dispatcher, Rose
If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
There is a record sellout.
Concert crowds are expected to leave
the coliseum around midnight.

I can hear those meters ticking now.
```

The yanked text stored in register **a** was put on the line after the current line (containing the cursor). The cursor goes to the first character in the first line of the put text. **Vi** remains in command mode.

Searching for Patterns in Last Line Mode

A pattern can be either a simple string (such as a word or an expression containing letters and numbers), or it can be a complex pattern containing special characters (which are also called metacharacters). Invoking last line mode from command mode, you can search forward or backward in a file for one or more occurrences of a pattern. The command format for searching forward follows.

/pattern

where:

/ is the delimiter that precedes the pattern that you are searching for. A search in a forward direction is implied.

The command format for searching backwards follows.

?pattern

where:

? indicates a pattern search in a backwards direction.

If **vi** finds the pattern, it will position the cursor on the first character of the pattern. Table 2-25 lists the commands that locate the next occurrence of a pattern.

Table 2-25 Commands to Search for the Next Occurrence of a Pattern

Command	Definition
n	Finds next occurrence of the pattern in the same direction as the original search.
N	Finds the previous occurrence of the pattern in the opposite direction of the original search.

If the pattern you are searching for contains a slash (/) or a question mark (?), be sure to escape it with a backslash (\) to prevent it from being interpreted as a search delimiter.

There are three editing options (see the section "Setting Up Your vi Environment" located in this chapter) that can control **vi**'s search behavior; they are:

- ignorecase.
- magic.
- wrapscan.

If the **ignorecase** option is set, then **vi** will not distinguish between upper- and lowercase characters when performing a search. If the **magic** option is set, you can use the full range of metacharacters for pattern searches. The full range is: ^, **$**, **\\**, **\\<** and **\\>**, *****, **.**, and **[ ]** (refer to Appendix A for a review of metacharacters). If the option is set to **nomagic**, then only a subset of the metacharacters will be recognized in pattern searches; it is: ^, **$**, , **\\<** and **\\>**.

If the **wrapscan** option is set, pattern searches cover the entire file. A search in a forward direction starts at the cursor position and proceeds to the end of the file. A search in a backward direction starts at the cursor position and proceeds to the beginning of the file, wrapping to the opposite end until the cursor position is reached again.

Metacharacters Used in a vi Pattern Search

Table 2-26 summarizes the metacharacters used in search patterns.

Table 2-26 Metacharacters Used in a vi Pattern Search

Metacharacter	Search Where?	Example
^*pattern*	Beginning of line.	**/^Dispatch**
pattern$	End of line.	**/Dispatch$**
pattern.	Any single character.	**.Dispatch**
*pattern**	No occurrence or multiple occurrences of preceding character.	**Dispatch***
\pattern	Escape meaning of metacharacter.	**\\.dog**
[*pattern*]	Surrounds choice of characters.	**[dD] [cC]**
pattern\\>	End of word.	**sling\\>**
\\<*pattern*	Beginning of word.	**\\<re**

Searching and Substituting Patterns

The substitute operation is a combination of the search and change operations. With the substitute command, you locate the desired pattern and replace it with a replacement pattern.

The format for the substitute command follows:

: [*address* [,*address*]] **s/***pattern***/***replacement-string***/**[**g**] [**c**]

A pattern is a regular expression delimited by slashes (**/ /**) by default. You can select any character to be used in delimiter pairs (such as ? ? or @ @). You should avoid using characters that are also used as metacharacters. Refer to Appendix A for a list of metacharacters. The replacement string must also be delimited by the same delimiters used for the search pattern.

Each part of the substitute command is described in the following sections.

The Address

By default, **vi** performs substitutions in the current line only. By using one or two addresses before the substitute command, you can make the substitution affect one line or a range of lines anywhere in the buffer. The special address symbol **%** can be used to specify all lines in the buffer. There are three ways to specify lines:

- By line number or symbol.
- By context (with regular-expression pattern matching).
- By offset.

Line Addressing

A line address can be the number of a line in the buffer; the symbol **$**, which means the last line in the buffer; or the symbol **.** (dot), which means the current line.

Context Addressing

A context address is a regular expression delimited by slashes (**//**) or question marks (??). It addresses the first line that matches the pattern, searching in a forward (**/**) direction or in a backward (?) direction. Offsets, discussed in the next paragraph, "Offset Addressing," also can be appended to a context address.

Information on regular-expression pattern matching is given in Appendix A.

Offset Addressing

You can specify lines relative to other addresses in this form:

address [**+** **−**]*n*

where:

address refers to a line or context address.

+ refers to after the addressed line.

− refers to before the addressed line.

n refers to the line number offset above or below the addressed line.

Table 2-27 gives examples of line addressing.

Table 2-27 Addressing Methods

Format	Example	Description
.	**.**	Addresses the current line.
$	**$**	Addresses the final line in the buffer.
%	**%**	Abbreviation for **1,$** (all lines).
.,$	**.,$**	Addresses all lines from the current cursor position to the end of the file.
+[+] [*n*]	**+3** **+++** **.+3**	Addresses the *n*th line following the current line. If line 100 is the current line, **+++** resets the new current line to 103.
−[−] [*n*]	**+3** **−−−** **.−3**	Addresses the *n*th line preceding the current line. If line 100 is the current line, **−−−** resets the new current line to 97.
x	**5**	Addresses one absolute line number represented by *x*.
x,y	**5,9**	Addresses a range of lines signified by two absolute line numbers represented as *x,y*.
'*reg*	**'a**	Addresses the line marked with the mark command (issued from last line or command mode) and an alphabetic register name.
''	**''**	Addresses the previous current line. Note that the symbol is formed by two single quotation marks.
?*pattern*?	**?^The?**	Addresses lines containing a pattern.
/*pattern*/,/*pattern*/	**/^The/,/Cabs/**	Addresses a range of lines identified by two patterns.
/*pattern*/,*x*	**/^The/,9**	Addresses a range of lines identified by a pattern and a single absolute line number, represented as *x*.

(continued)

Table 2-27 Addressing Methods

Format	Example	Description
x,/pattern/	**1,/^The/**	Addresses a range of lines identified by a single absolute line number, represented as *x*, and a pattern.
/pattern/, [.] [+ −]*n*	**/The/,.+20**	Addresses all lines from the pattern to 20 lines after the current line.
any-addressing-method!	**1,2!**	Addresses all lines except those specified in the range.

(concluded)

Search Patterns and Replacement Strings

The substitute operation is represented by the **s** command. A delimiter, such as **/**, follows the **s**, which introduces the search pattern. You can represent a pattern either as a literal pattern, such as the word "Dispatch," or by using regular-expression pattern matching. You terminate the search pattern with the same delimiter previously used to introduce the search pattern, such as **/**, followed by the replacement pattern such as "Widgets".

Table 2-28 lists several options that you can append to the substitute command.

Table 2-28 Substitute Options

Option	Description
g	**Vi** substitutes all occurrences of the addressed pattern in a line with the replacement string. By default, only the first occurrence of the addressed pattern in a line is substituted with the replacement string.
c	For each substitution, you are presented with a request to confirm the replacement. Each string to be replaced is underscored by caret (^) symbols and the cursor is positioned after the carets. By responding with "**y**," you confirm the substitution and the cursor moves to the next occurrence of a replacement pattern. Responding with a no answer, such as "**n**," you prevent the substitution. This procedure continues until you have confirmed or denied each replacement, or you interrupt the command with your interrupt key (typically Ctrl-C).

Behavior of Substitute Options with edcompatible Option

The substitute options, **g** and **c**, behave as toggles when the **edcompatible (ed)** editing option is set (refer to the section "Setting Up Your vi Environment"). If you set the edcompatible option, the first time you perform a substitution explicitly setting the **g** or **c** option, you are implicitly setting that option for subsequent search-and-replace commands. The next time you explicitly specify the same option, it has the implicit effect of turning off that option.

Substituting in the Current Line

The following sample text is used to show the effect of searching and substituting.

```
Dispatch to all the Cab Drivers on the Second Shift:
If you plan to be in the vicinity of
the Central City Coliseum tonight,
you are in for big business.
I can hear those meters ticking now.
```

An example of a simple search and replacement command follows:

:s/the/THE/ ↵

where:

s stands for substitute.

"`the`" is the pattern that you are searching for.

"`THE`" is the replacement string.

A delimiter (**/**) separates each field.

Since the cursor is positioned on the first line, only that line is searched for the identified pattern.

The result appears as follows.

```
Dispatch to all THE Cab Drivers on the Second Shift:
If you plan to be in the vicinity of
the Central City Coliseum tonight,
you are in for big business.
I can hear those meters ticking now.
```

The default is to substitute on the current line only. To search the entire buffer, use the percent sign (**%**) as the address (explained in the next section). Only the first occurrence of the search pattern "`the`" is substituted.

Substituting in the Entire Buffer (%)

Using the preceding sample file, you can now do a search and replacement affecting the entire buffer with the **%** command. An example follows.

:%s/the/THE/ ↵

The first line is scanned for the occurrence of the search pattern "`the`". The first occurrence of the search pattern is substituted with the replacement pattern "`THE`". The second occurrence of the search pattern on the first line, however, is not replaced. The **%** command recognizes only the first occurrence of the pattern in each line.

The final result appears as follows.

```
Dispatch to all THE Cab Drivers on the Second Shift:
If you plan to be in THE vicinity of
THE Central City Coliseum tonight,
you are in for big business.
I can hear those meters ticking now.
```

The **g** option recognizes multiple occurrences of a search pattern on a single line.

Global Substitution in a Single Line (g)

You can use the global option (**g**) to repeat a replacement for multiple occurrences of the pattern on a single line. For example, the following command would replace every occurrence of "`the`" in the first line of the sample file with "`THE`."

:1s/the/THE/g ↵

The final results appears as follows.

```
Dispatch to all THE Cab Drivers on THE second shift:
```

Confirming Substitutions (c)

You can substitute selected occurrences of a pattern replacement using the confirmation (**c**) option. An example follows:

:%s/THE/the/gc ↵

This example assumes that all occurrences of the search pattern "`THE`" are in uppercase.

```
Dispatch to THE Cab Drivers on THE Second Shift:
            ^^^□
```

Three carets underline the proposed first substitution. Global substitution stops temporarily awaiting your response. If you respond with **y** (for yes) and you press the New Line key, the substitution is performed and the process continues.

The next proposed substitution occurs as follows:

```
Dispatch to THE Cab Drivers on THE Second Shift:
                               ^^^□
```

Three carets underline the proposed substitution. Global substitution stops again awaiting your response. If your response is **y** (for yes) and you press the New Line key, another substitution is performed and the process continues.

An example of a **n** (no) response for the next proposed substitution follows.

```
If you plan to be in the vicinity of
                     ^^^□
```

The substitution is not performed and the cursor moves to the next occurrence of the proposed replacement string "THE".

```
THE Central City coliseum tonight,
^^^□
```

After all search patterns have been located and substitutions are confirmed, you are prompted:

```
3 substitutions on 2 lines
[Hit return to continue] ↵
```

You press the New Line key for a review of your substitutions. You will see:

```
[D]ispatch to all the Cab Drivers on the Second Shift:
If you plan to be in THE vicinity of
the Central City Coliseum tonight,
you are in for big business.
I can hear those meters ticking now.
```

You will notice that the first two substitutions of "the" for "THE" were confirmed. The third was not confirmed, and the fourth was confirmed.

Advanced Substitution

Table 2-29 summarizes the advanced substitution operators used for further manipulation of the replacement pattern.

Table 2-29 Advanced Substitution Operators

Operator	Definition
~	In the current substitution, repeats the replacement pattern from the previous substitution.
&	Saves search pattern and substitutes it in the replacement pattern.
\(\)	Surrounds (and thus tags) one or more search pattern(s) for substitution in the replacement pattern.
\u	Converts next single lowercase character in replacement pattern to its uppercase equivalent.
\U	Enables conversion of lowercase replacement pattern to its uppercase equivalent; the entire replacement pattern is converted unless **\e** or **\E** is encountered mid-pattern.
\l	Converts next single uppercase character in replacement pattern to its lowercase equivalent.
\L	Enables conversion of an uppercase replacement pattern to its lowercase equivalent; the entire replacement pattern is converted unless **\e** or **\E** is encountered mid-pattern.
\e	Ends case conversion for the remaining replacement pattern.
\E	Same as **\e**.

Repeating the Previous Replacement Pattern (~)

To repeat a replacement pattern for sequential search patterns, you can simply recall the previous replacement pattern using tilde (~). The format for repeating the previous replacement pattern follows:

:[*address*]s/*search-pattern*/~/

Sample text follows.

```
the Central City Coliseum tonight,
```

An example of a simple substitution command follows.

:s/the/tomorrow ↵

The result appears as follows.

```
tomorrow Central City Coliseum tonight,
```

Using the previous replacement pattern "tomorrow," you can repeat it in the next substitution using ~. An example follows:

:s/tonight/~/

The result appears as follows.

```
tomorrow Central City Coliseum tomorrow,
```

Substituting a Search Pattern in a Replacement Pattern (&)

This replacement operator is useful when you want to add text to the search pattern rather than substitute the search pattern with a replacement pattern. The ampersand (**&**) stores the text matched by the search pattern so that you can recall and place it in the replacement pattern. A search pattern can contain regular expression metacharacters.

The format for placing the search pattern in the replacement pattern follows:

:[*address*]**/***search-pattern***/***replacement-pattern***&/**

NOTE: The **&** can appear anywhere within the replacement pattern.

Sample text follows.

```
the Central City Coliseum tonight,
```

The following command adds a word to the sample text:

:s/tonight/EARLY &/ ↵

The result appears as follows.

```
the Central City Coliseum EARLY tonight.
```

Substituting with Tagged Replacement Patterns (\\(\\))

With the replacement pattern tag, you can search for and mark the text matched by as many as nine patterns. In the replacement pattern, you can then rearrange the order in which the tagged patterns occur.

The command format used for substituting with tagged replacement patterns follows:

:[*addr*]**s/\\(***tag-word1***\\)** **\\(***tag-word2***\\)**...**\\(***tag-word9***/\***tag-num* **\***tag-num*...**\***tag-num***/**

where:

\\(*tag-word***\\)** is formed by the escaped parentheses surrounding the pattern being tagged. A *tag-word* can contain regular expression metacharacters. A maximum of nine different search patterns can be tagged.

\*tag-num* causes the text matched by the corresponding tag to be substituted in the replacement pattern in the specified order. *Tag-num* is a single digit from **1** to **9**.

Sample text follows.

```
the Central City Coliseum tonight,
```

The following is an example of using tagged replacement patterns.

```
:s/\(Central\) \(City\)/\2 \1/ ↵
```

The first substring "`Central`" is tagged as 1; the second substring "`City`" is tagged as 2. The replacement tags cause the contents of locations 1 and 2 to be swapped in the standard output. Spacing in this command is important.

- Do not space within delimiters.
- Do not use a space to separate the final delimiter and the first replacement pattern.
- Do space once between replacement tags.

The result appears as follows.

```
the City Central Coliseum tonight,
```

You can see that locations 1 and 2 were swapped in standard output.

In addition to tagging patterns for swapping, you can also rearrange them with other replacement text. An example follows:

```
:%s/\(Central\) \(City\) \(Coliseum\)/Metro \2 \1 Events \3/ ↵
```

The result appears as follows.

```
the Metro City Central Events Coliseum tonight,
```

Converting Cases in the Replacement Pattern (\u \U \l \L)

You can use the case conversion operators to translate all or part of the replacement pattern to the alternate case. Within a replacement pattern, you can use multiple operators to alternate between cases.

The format for converting cases follows:

:[*address*]**s/***search-pattern***/***case-opreplacement-pattern***/**

NOTE: A case operator (*case-op*) can appear anywhere within the replacement pattern.

Sample text follows.

```
the Central City Coliseum tonight,
```

An example follows:

```
:s/tonight/\utomorrow/ ↵
```

The result appears as follows.

```
the Central City Coliseum Tomorrow,
```

Another example follows:

:s/tonight/\utom\Uorrow/ ↵

The result appears as follows.

```
the Central City Coliseum TomORROW,
```

The **\l** and **\L** operators work in a similar manner; uppercase characters are converted to lowercase characters instead.

Ending Case Conversion (\e \E)

You can use the **\e** and **\E** operators to end case conversion within the replacement pattern. You can use multiple case conversion operators within a replacement pattern.

The format for ending case conversion follows:

:[*address*]**s/***search-pattern***/***end-case-opreplacement-pattern***/**

NOTE: An end-case operator (*end-case-op*) can appear anywhere within the replacement pattern.

Sample text follows.

```
the Central City Coliseum tonight,
```

An example follows:

:s/tonight/\Utom\Eorrow/ ↵

The result appears as follows.

```
the Central City Coliseum TOMorrow,
```

Another example follows:

:s/tonight/\uto\Uni\eght/ ↵

The result appears as follows.

```
the Central City Coliseum TONIght,
```

Manipulating Files in Last Line Mode

From the last line, you can issue a variety of commands to operate on files. The commands perform these types of operations:

- Writing files.
- Reading files.
- Editing the buffer.
- Searching for a tag in other files.
- Exiting to the shell.

You issue these commands using this format:

:*command* [| *command* ...]

where:

The colon (**:**) command activates last line mode.

command operates on the current buffer. Some commands take an optional address. Refer to Table 2-27 for information on methods of addressing.

| separates optional multiple commands in last line mode. There is no limit on the number of commands you can enter in last line mode. Your input can wrap to the next line. As your text wraps to the next line, the top line of the screen will scroll up. If you use a global command, a comment, or a shell escape from last line mode, it must be the final command in the sequence.

You must press the New Line key to execute a command in last line mode.

Commands to Write Files

Table 2-30 lists the commands to write the buffer to a disk file. Precede each command with a colon (:), which signifies last line mode.

Table 2-30 Commands to Write the Buffer to a Disk File

Command	Description
[*addr*]w	Writes the entire current buffer (or optional lines represented as *addr*) to the disk file.
[*addr*]wq	Writes the current buffer (or optional lines represented as *addr*) to the disk file and quits the editor.
[*addr*]x	If you have made changes to the buffer and not written them, writes the buffer contents (or optional lines represented as *addr*) to the disk file, then quits the editor; same as **ZZ**.
[*addr*]w *file*	Writes the current buffer (or optional lines represented as *addr*) to a new disk file.
[*addr*]w! *file*	Overwrites the existing disk file with the current buffer (or optional lines represented as *addr*).
[*addr*] w>> *file*	Appends the buffer contents (or optional lines represented as *addr*) to the end of an existing file.
[*addr*] x *file*	If any changes have been made and not written, writes the buffer contents (or optional lines represented as *addr*) to the specified file and then quits the editor.
preserve	Saves the current buffer as if the system had crashed. Use for emergencies when a write command results in an error.

An example of writing an explicit number of lines from one file to another file using the [*addr*]w *filename* command is given using the following sample text.

```
Dispatch to all THE Cab Drivers on the Second Shift:
If you plan to be in THE vicinity of
THE Central City Coliseum tonight,
you are in for big business.
I can hear those meters ticking now.
```

This file is in the current buffer. To write this entire file or a subset of this file to another file, you would first find out the line numbers that specify the range of lines you want to write. You can set the line-numbering option in last line mode to show line numbers using this command:

```
:set nu ↵
```

The result appears as follows.

```
     1  Dispatch to all THE Cab Drivers on the Second Shift:
     2  If you plan to be in THE vicinity of
     3  THE Central City Coliseum tonight,
     4  you are in for big business.
     5  I can hear those meters ticking now.
```

A line number precedes each line in the current buffer

If you wanted to write the first through third lines to another file named **cabbies** while leaving the current buffer intact, you would issue this command from last line mode:

```
:1,3w cabbies ↵
"cabbies" [New file] 3 lines, 105 characters
```

The cursor returns to the first character in the current buffer. As another check that the file **cabbies** was written successfully, in last line mode you can temporarily exit to the shell and list the file in one step. You can issue the long list (**ls −l**) command to show the new file's access permissions (refer to *Using the DG/UX™ System* more information). The command you issue and the system response is given as follows:

```
:!ls cabbies ↵
-rw-rw-rw-   1  hank   general   105    May 10 6:26   cabbies
[Hit return to continue] □
```

Pressing the New Line key returns you to the current buffer.

To remove the line numbers in the current buffer, from last line mode you can issue this command:

```
:set nonu ↵
```

Commands to Read Files

Table 2-31 lists the commands used for retrieving copies of existing files for editing within **vi** without having to exit **vi** for the shell. Also, if you specified multiple files on the command line when you invoked **vi**, with these commands, you can retrieve one file after another within **vi** without having to exit to the shell.

Precede each command with a colon (**:**), which signifies last line mode.

Table 2-31 Commands to Read a File to the Current Buffer

Command	Description
e *file*	Retrieves a copy of the file for editing in the current buffer.
e!	Retrieves a copy of the file from the disk file in the current buffer for editing, discarding all changes made to current buffer.
e + *file-list*	Retrieves a copy of the first file in the *file-list* and positions the cursor on the last line in the buffer for editing.
e +*n file-list*	Retrieves a copy of the first file in the *file-list* and positions the cursor at line number *n* in the buffer for editing.
e #	Retrieves a copy of the previous file you edited (the one before the current file) and puts it in the buffer for editing.
n *file-list*	Specifies a new *file-list* and retrieves a copy of the first file and puts it in the buffer for editing.
n	Retrieves a copy of the next file in the *file-list* and puts it in the buffer for editing.
n *file*	Retrieves a copy of the specified file for editing; same as "**e** *file-list*" command.
n!	Retrieves a copy of the next file in the *file-list* without first writing the current buffer to disk file.
r *file*	Reads a copy of the disk file into the current buffer.
r !*cmd*	Exits to the shell to perform a command and reads the result into the current buffer.
*n***r** *file*	Reads a copy of the disk file into the current buffer after line *n*.
*n***r** !*cmd*	Exits to the shell to perform a command and reads the result into the current buffer at line *n*.
rew	Starts (rewinds) the current *file-list* over again.
rew!	Same as **rew** except that the contents of the current buffer are discarded.

A common task is to read a file into the current buffer at a specific location. An example of text in the current buffer follows:

```
Ski the summit
Ski 4 of the world's greatest ski areas on
1 lift package --- 8 different mountains ---
256 trails --- 53 lifts --- Located where
our airlines can take you!
What's next
```

With the cursor marking the line at which you want to read in the file, issue the read command from last line mode:

:r cabbie ↵

After you press the New Line key, the file is read into the current buffer. The result is shown as follows:

```
Ski the summit
Ski 4 of the world's greatest ski areas on
1 lift package --- 8 different mountains ---
Dispatch to all THE Cab Drivers on the Second Shift:
If you plan to be in THE vicinity of
THE Central City Coliseum tonight,
you are in for big business.
I can hear those meters ticking now.
256 trails --- 53 lifts --- Located where
our airlines can take you!
What's next
.
.
.
"cabbies" 5 lines, 194 characters
```

The cursor remains at the same line position and the contents of file **cabbies** is read into the current buffer at the cursor position. The remaining two lines of the original current buffer are moved down the screen to accommodate the new text. **Vi** displays a status message on the last line reporting the filename and the number of lines and characters it contains.

To undo the result of the read command, you can issue the **u** command from command mode; **vi** then returns the buffer to its previous state (without the contents of the file read into the current buffer).

Commands to Edit the Buffer

Table 2-32 lists the commands to edit the buffer in last line mode, which is invoked by using the colon (**:**) command. Some of these editing functions can also be performed in **vi**'s command mode.

Several of these commands take the optional final argument **p**[*offset*].

where:

p prints each line changed by an editing operation (after the operation has been performed), and the number of lines changed by an editing operation. It then refreshes the screen, positioning that line in the middle of the display as the new current line.

[*offset*] is an optional argument to **p** that resets the current line. It represents a relative motion from the current line. For example, one or more plus (**+**) or minus (**−**) signs will move the cursor that many times in a forward or backward direction relative to the current line. Both of these representations, **l+++** and **l3+**, move the current line forward three lines.

For some buffer editing commands, addresses are expressed in two explicit ways: *addr* and *single-addr*.

addr designates a range of lines.

single-addr designates a single line.

In some cases, *from-addr* and *to-addr* are used to tell you the locations in your file where text will be edited.

For some commands, you can specify *n* lines from the current line as an alternative to *addr* (as previously discussed).

Table 2-32 Commands to Edit the Buffer

Command	Description
address	Displays a window of text with the specified address in the middle of the screen. The current line is the default address.
args	Prints the current list of files being edited. The current argument is surrounded by brackets.
[*from-addr*] **co** *to-addr* [[**p**] [*offset*]]	Copies the *from-addr* line(s) to the line following the *to-addr* in the buffer. The current line is the default *from-addr*. A synonym for **co** is **t**.
[*addr*] **d** [*buffer-name*] [*n*] [[**p**] [*offset*]]	Deletes the addressed line(s) from the buffer. The current line is the default address. If you give a lowercase *buffer-name*, the command copies deleted lines to it; for an uppercase *buffer-name*, the command appends deleted lines. Deleted text is saved in a general buffer if you do not explicitly specify one. Buffer contents are restored with the put command (see the **pu** command further in this table).
file [*filename*]	The first form (without a *filename* argument) prints the current filename, whether it has been modified since the last write command, whether it is read only, the current line, the number of lines in the buffer, and the percentage of the file that precedes the current line. The second form (with a *filename* argument) changes the disk filename to *filename* without changing the contents of the current buffer.
[*addr*] **g/***pattern***/***cmd* [| *cmd* ...]	Globally searches the addressed lines for occurrences of the search *pattern*. Then these lines are operated on by the specified command(s). The entire file is the default address. You can separate multiple commands with a pipe symbol (|).

(continued)

Table 2-32 Commands to Edit the Buffer

Command	Description
[*addr*] **g!/***pattern***/***cmd* [| *cmd* ...]	Within the range of lines addressed, searches for all lines that do not match the search pattern. Then these lines are operated on by the specified command(s). The entire file is the default address. You can separate multiple commands with a pipe symbol (|). A synonym for **g!** is **v**.
[*addr*] **j** [*n*] [[**p**] [*offset*]]	Links (or joins) the addressed lines onto a single line. The current line through the next line are joined by default. White space is adjusted automatically. At least one space is inserted between joined words; two for sentences (separated by a period .). White space is deleted from the beginning of joined lines, if it exists.
[*addr*] **j!** [*n*] [[**p**] [*offset*]]	A variation of **j**; links (or joins) the addressed lines onto a single line with no white space adjustment. Links the contents of lines verbatim. The current line through the next line are joined by default.
[*single-addr*] **mark** *buffer-name*	Marks a single line for subsequent positioning or editing. The current line is the default address. A synonym for **mark** is **k**.
[*addr*] **l** [*n*] [[**p**] [*offset*]]	Temporarily prints (or lists) the addressed lines, displaying tabs and new-lines as **^I** and **$**, respectively, at the bottom of the screen. You then press the New Line key to return the screen to its previous state. The current line is the default *addr*. The final line printed becomes the new current line.
[*from-addr*] **m** *to-addr*	Repositions (moves) the text designated by *from-addr* to the line following *to-addr*. The current line is the default *from-addr*. The first line of the moved text becomes the new current line.

(continued)

Table 2-32 Commands to Edit the Buffer

Command	Description
[*addr*] **nu** [*n*] [[**p**] [*offset*]]	Temporarily prints (or numbers) each addressed line with its corresponding buffer line number at the bottom of the screen. You then press the New Line key to return the screen to its previous state. The current line is the default address. The last line printed becomes the new current line. A synonym for **nu** is **#**.
[*addr*] **p** [*n*]	Temporarily displays (or prints) each addressed line on the screen. The current line is the default address. You then press the New Line key to return the screen to its previous state. The final line printed becomes the new current line. A synonym for **p** is **P**.
[*single-addr*] **pu** [*buffer-name*]	Places (or puts) previously deleted or yanked text (see the **d** and **ya** commands, respectively) in the buffer following the specified address. The current line is the default address. The default *buffer-name* is the general buffer where the text from the immediately preceding operation was deleted or yanked to. A specific *buffer-name* restores explicitly saved text to the addressed buffer location. The final line of the put text becomes the new current line.
u	Reverses (or undoes) the changes made to the buffer by the previous editing command, including the global command (**g** or **g!**), which can affect more than one line. Issuing the **u** command a second time reinstates the change; thus, it behaves as a toggle.
ve	Reports the current version of the **vi** editor.

(continued)

Table 2-32 Commands to Edit the Buffer

Command	Description
[*addr*] **ya** [*buffer-name*] [*n*]	Duplicates (or yanks) the addressed lines, storing them in a buffer for subsequent retrieval via the **pu** (put) command. The current line is the default address. If you do not specify a buffer, the yanked text is saved in a general buffer. Alternatively, you can save yanked text in an explicitly named buffer. If you give a lowercase buffer name, this command copies yanked lines to it; for an uppercase buffer name, this command appends yanked text to it. The current line does not change.
[*addr*] **w** **!***command*	Sends (or writes) the addressed lines as standard input to a *command* that is executed from a subshell. The entire file is the default address. Note the difference between this command and the **w!** command (covered in the section on "Commands to Write Files"), which overwrites the existing disk file with the current buffer.
[*single-addr* **+ 1**] **z** [*n*]	Prints a range of lines on the screen, extending from the line following the specified address through a line number which is defined as half the value of the window editing option (see section on "Options for Slow Terminals"). For example, if window=23, the command, **:5z**, would cause buffer lines 6-16 and the status line reserved at the bottom of the screen to be displayed. Line 2 is the current address. The last line of the range of text displayed becomes the new current line.

(continued)

Table 2-32 Commands to Edit the Buffer

Command	Description
[*single-addr*] **z** [*type*] [*n*]	Temporarily prints a window of text on the screen, positioning the specified address according to *type* or *n*. The current line is the default address. *type* positions the specified address in the window: default Puts address at top of screen. . Puts address at middle of screen. – Puts address at bottom of screen. ! Scrolls full screen – 1 line up so address is off top of window. ^ Scrolls screen back so address is just off bottom of screen. + Scrolls screen forward so address is just off top of screen. = Puts address at center of screen with a row of hyphens above and below addressed line. The amount of text scrolled in the window is equal to the double value of the scroll editing option (see section on "Options for Slow Terminals"). Alternatively, you can specify a window of text whose length is determined by an absolute *n* value. If no *n* value is given, 1 line is assumed by default. The last line in the window becomes the new current line.
[*addr*] **>** [**>** ...] [*n*] [[**p**] [*offset*]] [*addr*] **<** [**<** ...] [*n*] [[**p**] [*offset*]]	The first form shifts text on the addressed line to the right (**>**); the second form, to the left (**<**). The amount of space shifted depends on the value of the shiftwidth editing option (see section on "Programming and Debugging Options"). The current line is the default address. You can specify more than one shift by using multiple shift operators (such as **>>>**). The final line of the addressed text that is shifted becomes the new current line.
$=	Reports the number of the last line in the current buffer.

(concluded)

Searching for a Tag in Other Files

A tag identifies a function or optionally a typedef in a file containing program code, which is typically written in the C, Pascal, and FORTRAN languages. Before you can use the tag search facility, you must have a tag file named **tags** in your current directory that you produced using the program **ctags**. Refer to *Using the DG/UX™ System* for information on creating a tag file and understanding its contents.

NOTE: In addition to generating a tag file for a programming applications, you can also manually generate a tag file for text applications using a text editor such as **vi**. Regardless of the method used for generating such a file, you can use the **vi** commands for tag searches in other files.

Table 2-33 lists the tag commands that you can issue in last line mode (by using the colon (**:**) command) to search for a tag label in another file.

Table 2-33 Commands to Search for a Tag in Other Files

Command	Description
tag *label*	Given that a tags file exists in the correct format, **vi** clears the buffer and positions the cursor on the first nonblank character in the line containing the tag. If the tag is in a file different than the one in the current buffer, you must write the current buffer to a disk file. If the autowrite option (**aw**) is set (refer to the section "Setting Up Your vi Environment" in this chapter), then the current buffer is written for you before switching files.
tag! *label*	Same as the previous command except that the changes made to the current buffer are dismissed first.

Commands to Exit to the Shell

With the following commands, you can exit **vi** temporarily (without having to write and quit) and enter the shell to execute a command, and then return to the current file to resume your work session. The current buffer remains intact during the shell escape. Precede each command with a colon (**:**), which signifies last line mode. Table 2-34 lists the shell exit commands.

Table 2-34 Commands to Exit to the Shell

Command	Description
! *command*	Exits to a subshell and runs a command immediately. As soon as the command completes its execution, you are prompted with this message: `[Hit return to continue]`. Pressing the New Line key returns you to **vi**. You can use the !! command to recall and re-execute the previous shell command you issued, regardless of the shell used.
[*addr*] ! *command*	The text lines defined by the address are taken as standard input to the command executed in the subshell. The resulting output replaces the input lines. For example, the command **1,5 ! grep "dog"** causes the **grep** command to search for the pattern "dog" in lines 1–5. Only the lines containing an occurrence of the pattern are written back to the input file. The remaining lines not containing the pattern are deleted.
sh	Transfers control to a subshell of your default shell: the Bourne shell or the C shell. You can use Ctrl-D or the **exit** command to return control to the **vi** editing session.
stop	C shell only; Transfers control to the parent shell of the current **vi** process. One advantage of using **stop** instead of **sh** is that with **stop**, the commands you enter at the shell are captured in the parent shell's history list. Also, it prevents accidental multiple **vi** sessions, and it uses fewer system resources than the **sh** command. Enter the **fg** command from the shell to return control to the suspended **vi** process.

Once you are in a subshell, you may forget that **vi** is suspended. In fact, you can invoke **vi** with the current filename again and continue working. However, you will not be working on the version of the file in the buffer; instead, you are working on a copy of the most recent file written to disk. Always terminate a subshell to return to the current **vi** session instead of starting more editing sessions. You can terminate the subshell using either of these methods:

```
$ exit ↵
```

or

```
$ Ctrl-D
```

If you are using the C shell, type the **fg** command to return to the current **vi** session.

Setting Up Your vi Environment

You can customize your **vi** environment to suit your personal tastes by turning on an option or by assigning a specific value to an option. Such options include right margin set-up, error and status message display, and definition of window size.

Also, you can define macros to represent a sequence of operations. Refer to the section "Writing and Using Macros" in this chapter for more information.

How to Set Options

There are three command formats used to set an editing option. With all of these forms, you can set multiple options in one command. Issue each of these command formats in last line mode (from command mode, use the colon (**:**) command).

Turn an Option On or Off

The command formats for turning an option on and off follow:

> **set** *option-name*
> **set no***option-name*

These commands are terminated with a new-line.

Examples follow:

> **:set redraw** ↵
> **:set noredraw** ↵

In the first example, the redraw option is enabled. It causes the terminal screen to be updated after each editing operation. Otherwise, updating is postponed until you press the Escape key. This option is useful if you have a slow terminal. The option is disabled in the second example.

Set an Option to a Number Value

The command format for setting an option to a number value follows:

> **set** *option***=***number*

An example follows:

> **:set wm=10** ↵

The wrapmargin (**wm**) option generates an automatic new line between words at or after the defined column position from the right side of the screen.

Set an Option to a String Value

The command format for setting an option to a string value follows:

set *option*=*string*

An example follows:

:set term=vt100 ↵

The **set** command is used to assign the value of **vt100** to the variable **term**.

Where to Set Options

There are three methods for setting these options:

- **EXINIT** variable.
- **.exrc** file.
- Last line mode.

Each time you log in to your system, a specific initialization sequence is used. The order of processing follows this sequence.

- If an **EXINIT** variable is set, it is initialized.
- If no **EXINIT** variable is set, then the **.exrc** file in the log-in directory is initialized.
- If you set up multiple **.exrc** files in several directories, the one in the current directory is initialized.

EXINIT Variable

This method will start **vi** faster than the other two. If you set it on the command line, it will be effective for the current log-in session only. You can also set this variable in the appropriate setup file—**.profile** for Bourne shell users and **.login** for C shell users. If you choose to set it in a setup file, it will be in effect automatically each time you log in. If the **EXINIT** variable is present, it will override whatever settings you have made in the **.exrc** file in your home directory, but not the one set up in the current directory, if you have multiple **.exrc** files.

The command format varies between the Bourne shell and the C shell. Each form is given.

Bourne shell:

```
EXINIT='set option option option ...'
export EXINIT
```

C shell:

```
setenv EXINIT 'set option option option ...'
```

Examples of each follow.

Bourne shell:

```
EXINIT='set wm=20 smd redraw' ↵
export EXINIT ↵
```

C shell:

```
setenv EXINIT 'set wm=20 smd redraw' ↵
```

As another alternative, you can set up the **EXINIT** environment variable in your setup file—**.profile** for the Bourne shell and **.login** for the C shell.

.exrc File

You can create a file named **.exrc** in your log-in directory for your **set** commands. The options you set in this file will be in effect for all editing sessions for any shell. You can also create multiple **.exrc** files, placing them at different subdirectory levels. The **.exrc** file in the current directory will override the **.exrc** file in the log-in directory. An example of the contents of an **.exrc** file follows:

```
set wm=20
set smd
set redraw
```

Or you can put multiple options on a line, separating each option with any amount of space.

If you put macro definitions (single-character commands that represent a sequence of operations) in the **.exrc** file, you will need to separate each key map and separate the first key map from the options with a column separator (|). More information on macros is in a later section in this chapter. An example follows:

```
set wm=20 set smd set redraw | map! z italic | ab mm Missoula, Montana ↵
```

In the preceding example, you set three **vi** editing options, mapped the **z** key to the `\italic` string, and mapped the abbreviation macro name `mm` to the string `Missoula, Montana`. More information on mapping is given in the section "Writing and Using Macros" later in this chapter.

Last Line Mode

You can set options in last line mode that remain in effect only for the current editing session. When you terminate **vi**, these settings are deleted. The same format used for setting options in the **.exrc** file is used in last line mode.

Displaying the Current Options Set

You can display the current settings for either of the following:

- Specific option.
- Changed option.
- All options.

Display the Value of a Specific Option

For options that take number or string values, you can find find out the current value you set during an editing session with the following command format:

set *option-name*

An example follows:

```
:set wm ↵
wrapmargin=5
```

From the last line, you request the value of the wrapmargin (**wm**) option. The option and current value is reported in the last line and the cursor returns to its previous position on the terminal screen.

Display Values of Changed Options

A changed value is one you set that is different from the default. You can get a report of the changed options using this format:

set

An example follows:

```
:set ↵
autoindent autowrite redraw report=2 scroll=3 shiftwidth=5
showmode term=vt100 wrapmargin=5
[Hit return to continue]□
```

From the last line, you request the values of all options whose values you have changed from the default. The options and current values are reported at the bottom of the screen. To remove the display, press the New Line key.

Display Values for All Options

The following command format displays the current values, changed or default, of each **vi** option.

set all

Figure 2-3 shows an example of a typical editing options display.

```
: set all ↵
autoindent          nonumber                       showmode
autoprint           nonovice                       noslowopen
autowrite           nooptimize                     tabstop=8
nobeautify          paragraphs=IPLPPPQPP LIpplpipnpb taglength=0
directory=/tmp      prompt                         tags=tags /usr/lib/tags
noedcompatible      noreadonly                     term=vt100
noerrorbells        redraw                         noterse
flash               remap                          timeout
hardtabs=8          report=2                       ttytype=vt100
noignorecase        scroll=3                       warn
nolisp              sections=NHSHH HUuhsh+c        window=23
nolist              shell=/bin/sh                  wrapscan
magic               shiftwidth=5                   wrapmargin=5
mesg                noshowmatch                    nowriteany
[Hit return to continue]□
```

Figure 2-3 Typical Editing Options Display

From the last line, you request the values of all options having default and changed values. All options and current values are reported at the bottom of the screen. To remove the display from the screen, you press the New Line key.

The Options

The options are divided into these groups:

- Editing.
- Programming and debugging.
- Slow terminals and those operating via a modem.
- DTK (Documenter's Tool Kit) applications.

Editing Options

Table 2-35 lists the editing options.

Table 2-35 Editing Options

Name	Abbrev	Default	Description
autoprint	**ap**	**ap**	Option is on or off; displays changed lines to show the result of an editing operation performed on the last line.
autowrite	**aw**	**noaw**	Option is on or off; if set to **aw**, before you switch to editing a different file, exit the editor, or issue a shell escape command, the current buffer is written to the current disk file if the buffer contents have changed since the last write (see the section "Manipulating Files in Last Line Mode").
beautify	**bf**	**nobf**	Option is on or off; when set to **bf**, when reading a file into the current buffer (see the section "Reading Files"), removes all control characters from the display (or "beautifies" it) with the exception of these control sequences: Ctrl-I (Tab) Ctrl-J (New Line) Ctrl-L (Formfeed)
directory	**dir**	**dir=/tmp**	Takes string value; defines the default temporary file used as additional buffer if your working buffer overflows.
edcompatible	**ed**	**noed**	Option is on or off; if set to **edcompatible**, then the last line mode substitution command is affected. When the **g** (global) or **c** (confirm) option (or both) is set, all subsequent substitute commands set these options implicitly. These options remain in effect until you explicitly specify them as arguments to the substitute command. In the following command, the options are turned on explicitly: **:%s/the/THE/gc** ↵

(continued)

Table 2-35 Editing Options

Name	Abbrev	Default	Description
edcompatible	**ed**	**noed**	In the next example, no options are set explicitly but the implicit effects remain: **:1,80s/^Dispatch/Message/ ↵** To turn off the **g** and **c** options, you must set them again explicitly. **:%s/Cab/Taxi/gc ↵**
error bell	**eb**	**noeb**	Option is on or off; beeps to signal an error on terminals that cannot flash.
flash	**fl**	**nofl**	Option is on or off; for some terminals, error conditions are signalled by a brief flash on the screen rather than a beep.
hardtabs	**ht**	**ht=8**	Takes numeric value; matches **vi** tab settings to the hardware tab settings of your terminal or the software tab settings. This option is useful only if your terminal uses tab settings other than every 8 spaces.
ignorecase	**ic**	**noic**	Option is on or off. If set to **ic**, ignores case when searching for regular expressions (see "Searching for Regular Expressions"). There is no difference between upper- and lowercase.
list	**list**	**nolist**	Option is on or off; displays normally invisible control characters — **^I** for a tab and **$** for a new-line.
magic	**magic**	**magic**	Option is on or off; if **nomagic** is set, ignores metacharacters used for regular-expression pattern searching except: **^**, **$**, \, **\\<**, and **\\>**. These metacharacters are ignored: *****, **.**, **&**, **~**, and **[ ]**. To reinstate the recognition of all metacharacters even with **nomagic** set, precede each metacharacter with a backslash (\). Also, refer to Appendix A for information on "Regular Expressions."
message	**mesg**	**mesg**	Option is on or off; enables or disables display of system messages during an editing session. Messages are sent with the **write**(1), **wall**(1M), or **rwall**(1M) commands.

(continued)

Table 2-35 Editing Options

Name	Abbrev	Default	Description
novice	**novice**	**nonovice**	Option is on or off; useful for C shell only. It should be set in the **EXINIT** environment variable or a **.exrc** file; it should not be set in **vi** last line mode. If set to **nonovice**, you can use Ctrl-Z or the downarrow (↓) to suspend **vi**. Control is sent to the shell. To resume **vi** and put it in the foreground, you can issue the **fg** command with the job number and the screen is refreshed with the current buffer. If **novice** is set, the suspend function is disabled and you can use the downarrow as a cursor scan key.
number	**nu**	**nonu**	Option is on or off; numbers each line in the file.
readonly	**ro**	**noro**	Option is on or off; the file is set to read-only mode; you can view, but not write to the file.
remap	**remap**	**remap**	Option is on or off; if **remap** is set, recognizes multiple mappings in a single macro (refer to the section "Writing and Using Macros" later in this chapter). For example, if a macro named **x** is mapped to **y** and **y** is mapped to **z**, then with mapping on, **x** maps to **z**. If off, **x** maps to **y** only.
report	**report**	**report=5**	Takes numeric value; **vi** reports the number of lines changed, deleted, or yanked by your last command if the number of lines affected is greater than the *report* value.
shell	**sh**	**sh=$SHELL**	Takes string value; defines the name of the shell that is executed when you issue a command in the form — **:!**. By default, this shell is defined by the **$shell** or **$SHELL** variable for the Bourne shell and the C shell, respectively (see chapters 5 and 6 for more information). An example follows: **sh=/bin/csh ↵**

(continued)

Table 2-35 Editing Options

Name	Abbrev	Default	Description

showmode	**smd**	**nosmd**	Option is on or off; displays the appropriate mode on the status line:
			`APPEND MODE`
			`SUBSTITUTE MODE`
			`CHANGE MODE`
			`REPLACE MODE`
			`OPEN MODE`
			`INSERT MODE`
			`INPUT MODE`

tabstop	**ts**	**ts=8**	Takes numeric value; specifies the interval between tab stops.
term	**term**	**term=$TERM**	Takes string value; defines your terminal type for the system from the list of legal terminal-type values. The value you specify must be consistent with the terminal type defined in either the **TERM** environment variable or the appropriate setup file—**.profile** for Bourne shell users and **.login** for C shell users. (Refer to *Using the DG/UX™ System* for more information on the terminal type). An example follows: **term=vt100** ↵
terse	**terse**	**noterse**	Option is on or off; if set to **terse**, error messages are shortened.
timeout	**to**	**noto**	If **timeout** is on, when you use a macro name exceeding one character (see the section "Writing and Using Macros" later in this manual), you must enter at least a character per second or the name will be interpreted as regular text.

(continued)

Table 2-35 Editing Options

Name	Abbrev	Default	Description
warning	**warn**	**warn**	Option is on or off; issues or inhibits warning that file hasn't been written to disk. If this option is set to **warn**, and you try to escape to the shell using the **:sh** or **:!** commands, this message will be displayed: `"[No write since last change]"`. This message is a warning that the information you are editing is not consistent with the disk file.
wrapmargin	**wm**	**wm=0**	Takes numeric value; generates automatic new line between words at or after the defined column position from right side of screen. If set to 0, automatic wrap is not performed.
wrapscan	**ws**	**ws**	Option is on or off; if **ws** is set, for regular-expression pattern search operations, the search starts at the cursor position, goes to the end of the file, and then to the top of the file until the cursor position is reached. If **ws** is off, the search is restricted to the cursor position to the end of the file.

(concluded)

Programming and Debugging Options

Table 2-36 lists the programming and debugging options.

Table 2-36 Programming and Debugging Options

Name	Abbrev	Default	Description
autoindent	**ai**	**noai**	Option is on or off; when set to **ai**, when **vi** is in input mode, each new line begins automatically in the same column as the column beginning the previous line. To move left of that column, use Ctrl-D; to move right, use a space or tab. This option is particularly useful for entering programming code that uses statement and construct nesting.
lisp	**lisp**	**nolisp**	Option is on or off; recognizes LISP delimiters for automatic indentation. If **lisp** is set, you can indent **lisp** code appropriately. Also, it modifies these symbols, which have a special meaning for **lisp**: **( ), { }, [[,** and **]]**. Also, it enables the **=** (formatted print) operator when using S-expressions. See the **showmatch** option in this table.
modelines	**ml**	**noml**	Option is on or off; if **ml** is set, the first and last five lines of each input file are checked for embedded **ex** commands (see the **ex**(1) man page in the *User's Reference for the DG/UX™ System*). If such commands are found, they are executed. Such a command must be preceded by **:ex** or **:vi**, and must be followed by **:**. This option can be set to apply to specific program source files, such as a tags file, by embedding the command in a comment in the source file.
shiftwidth	**sw**	**sw=8**	Takes numeric value; useful only when **autoindent** is on. It specifies the software tab distance for the tab stops for the Ctrl-D, space, or tab commands, necessary for tabbing in a backwards direction (to the left) with the **autoindent** option (located in this table).

(continued)

Table 2-36 Programming and Debugging Options

Name	Abbrev	Default	Description
showmatch	**sm**	**nosm**	Option is on or off; when **sm** is set and **vi** is in input mode, when you type a closing brace (}) or parenthesis ()) that matches an opening brace ({) or parenthesis ()) on the same screen, the cursor jumps to that matching character for a second, then back, and you can continue entering text.
taglength	**tl**	**tl=0**	Takes numeric value; if the length is nonzero, the tag name is significant to this many characters. If the length is 0, all characters are significant. Refer to *Using the DG/UX™ System* for more information.
tags	**tag**	**tag=tags /usr/lib/tags**	Takes string value; defines the name(s) of one or more tags files (tags files do not have to be produced by **ctags** and they do not have to be named **tags**). Each space-separated word in the **tags** string is the path of a tags file, which will be searched sequentially. The default **tags** string, **tags /usr/lib/tags**, means to first search the tags file named **tags** in the current directory. If no match is found in that file, then the tags file, **/usr/lib/tags** will be searched (refer to the section on **ctags** in *Using the DG/UX™ System*). As an alternative to the default, you can set up different tags files throughout your directory system and specify that they are to be searched by **vi**. The format follows: **set tags=/*filepath*/tags /*filepath*/tags** ⮐
writeany	**wa**	**nowa**	Option is on or off; if **wa** is set, turns off the checks made by **vi** when you want to write a file. If **nowa** is set, turns on the checks. You must use **w!** to override the checks. Refer to the section "Commands to Write Files" for more information.

(concluded)

Options for Slow Terminals

Table 2-37 lists the options that are useful for terminals connected to slow communications lines or those lacking "smart" terminal capabilities (such as to redraw your screen with each editing command). All of these options are terminal- and speed-dependent.

Table 2-37 Options for Slow Terminals

Name	Abbrev	Description
redraw	**redraw**	Option is on or off; when **redraw** is on, redraws (or updates) the entire screen after each editing operation. Otherwise, updating is postponed until you press the Escape key. The redraw attribute is in effect automatically for powerful "smart" terminals operating at a high speed. When operating at a low baud rate, setting this option to **noredraw** accelerates transmission speed at the expense of the appearance of overwriting existing text.
optimize	**optimize**	Option is on or off; particularly helpful when operating via modem. If **optimize** is set, the screen refresh rate is accelerated. If your text includes leading spaces on a line and your terminal interprets the new-line character as a line feed (without changing the column), then leading space characters can be avoided for that line.
scroll	**scr**	Takes numeric value; defines the number of lines to scroll when you use Ctrl-D (scroll down) and Ctrl-U (scroll up). If the window is 23 lines, then the scroll size is 11. By default, scroll size is half the window size. Refer to the section "Scrolling and Paging Text Through the Current Window" in this chapter for more information.
slowopen	**slow**	Option is on or off; for some terminals at slow baud rates, **vi** will not open a blank line on the screen when you are entering a new line of text (such as when you use the **o** command). Your editing actions will appear to overwrite existing text. With **slowopen** off, your screen is refreshed automatically. When operating via modem or at low baud rates, setting this option to off will accelerate transmission speed at the expense of a noncurrent screen.

(continued)

Table 2-37 Options for Slow Terminals

Name	**Abbrev**	**Description**
window	**wi**	Takes numeric value; this option is terminal dependent. If you do not set the option, window size is set according to the baud rate at which your terminal operates: **wi=***screen-size* **− 1** greater than 1200 baud **wi=16** 1200 baud **wi=8** less than 1200 baud You can set the appropriate one explicitly. As an alternative, you can declare the desired window size according to the baud rate at which you are operating. The following sizes correspond with baud rate: **w300** less than 1200 baud **w1200** 1200 baud **w9600** above 1200 baud Examples follow: **:set w300 = 15** ↵ **:set w1200 = 23** ↵ When operating at 300 baud, the screen size will be 15 lines; when operating at 1200 baud, the screen size will be 23 lines.

(concluded)

DTK Options

Table 2-38 lists the options that are useful if you use the **nroff** and **troff** macros.

Table 2-38 DTK Options

Name	Default	Description
paragraphs	**para=IPLPPPQPP LIpplpipnpbp**	Takes string value; each pair is an **nroff/troff** command that signifies a paragraph delimiter, which is recognized by the **vi** paragraph text objects, **{** and **}**. Refer to the section "How vi Relates to the Document Formatter: DTK" for more information. These defaults are basic **mm**, **me**, and **ms** paragraph macros. You can replace the defaults with any macros you have written. If you have a single-letter macro, such as **.p**, append a backslash (\\) and a space. The following example shows how to replace the defaults with **.p** and **.PP**. **:set paragraphs=p\\ PP** ↵
sections	**sect=NHSHH HUuhsh+c**	Takes string value; each pair is an **nroff/troff** command that signifies a section delimiter, which is recognized by the **vi** paragraph text objects, [and]. Refer to the section titled "How vi Relates to the Document Formatter: DTK" for more information. These defaults are basic **mm**, **me**, and **ms** section macros. You can replace the defaults with any macros you have written. If you have a single-letter macro, such as **.s**, append a backslash (\\) and a space. The following example shows how to replace the defaults with the macros **.s** and **.SH**. **:set paragraphs=s\\ SH** ↵

The DG/UX system supports only the **mm** macro set.

Table 2-39 contains a list of the relevant macros contained in this macro set:

Table 2-39 mm Macro Set Commands

Command	Definition
.P	New paragraph
.LI	List item
.H	Numbered paragraph
.HU	Unnumbered heading
.bp	Begin new page

You may want to edit the paragraphs and sections options to show only the **mm** macro commands.

Writing and Using Macros

A macro is a single-character (preferably) command name that represents a sequence of operations. To save keystrokes and time, you can set up frequently used commands or words into macros. There are two types of macros:

- Key map macro.
- Abbreviation macro.

You can assign a series of commands to one or more letter keys (for example, **m**) or to a function key (for example, F1). You can write macros to be used in command mode or input mode. For command mode, you will use some of the **vi** editing commands to do such operations as append, insert, delete, or change.

A variation of a key map for input mode is the abbreviation macro, which expands a macro name into a corresponding word. The difference between the key map used in input mode and the abbreviation macro is that the latter is used only when it is entered as a word by itself.

Where to Set Up Macros

There are three methods for setting up macros:

- **EXINIT** variable.
- **.exrc** file.
- Last line mode.

Refer to the section "Setting Up Your vi Environment" for more information.

Using Key Maps

You can define key maps to use in both **vi** command mode and insert mode. The syntax used for defining such key maps follows.

The following command format defines a macro for use in **vi** command mode:

> **map** *macro-name definition*

The following command format defines a macro for use in **vi** insert mode:

> **map**[**!**] *macro-name definition*

where:

map is the keyword for command mode macros; **map!** for insert mode macros.

macro-name labels the set of commands that are invoked each time you issue the macro. Although one character is preferable, a macro name can contain as many as 10 characters. If a macro name contains more than one character, you will have to type at least one character a second unless the **notimeout** option is set. See the section "Setting Up Your vi Environment." Macro names should not contain characters having a special meaning in **vi** command mode such as **i** (insert), **a** (append), and **o** (open). There is no restriction, however, on using characters that have meaning in last line mode, such as **q** (quit) and * (a regular-expression metacharacter) in macro names. The following keys are available for user-defined macro names:

g	q	v	K	V
Ctrl-A	Ctrl-O	Ctrl-T	*	\

The availability of specific keys for macro names depends on your terminal type. Make sure a given key is not defined before you use it for a macro. To avoid key map duplication, you can check the macros currently set with the **map** command (see the next section). Also, do not use **vi** commands in new macros. Alternatively, a macro name can also be a function key. You are limited to using function keys F1 to F10. The function keys on your keyboard may not be labelled as such. Regardless of labelling, the function keys are usually located on the top row of the keyboard or in a group to one side of the alphanumeric keys. You specify a function key as ***#n***, where *n* is **1** through **10**.

A macro *definition* can contain any number of commands, which can add up to 100 characters. If you use multiple commands in a macro definition, it is not necessary to space between them. Table 2-40 lists the control characters that you will need to precede with a single escape character, represented as a Ctrl-V.

Table 2-40 Escaping Control Codes in Macros

Control Key	Keystroke
Escape	Ctrl-V
New Line	Ctrl-J
Carriage return	Ctrl-M
Escape key	Ctrl-[

Setting Up a Command Mode Macro

A macro to perform the write and quit operation in last line mode follows. You type the following macro definition:

:map q :wq<Ctrl-V> ↵ ↵

The result of what you type will look like this:

```
:map q :wq^M
```

You assign the **q** key to this function, the write and quit (**:wq**) command, followed by one escape character. The Ctrl-V is displayed as a single caret (^) that escapes the next character you type, which is the new-line. The new-line is displayed as an "M". The first new-line is part of the definition; the second new-line ends the command sequence and returns control to **vi**.

To ensure that the macro definition has taken effect, you can type the **map** command from last line mode. The current macro names are listed.

```
:map ↵
up       ^W       k
q        q        :wq
left     ^Y       h
right    ^X       l
home     ^H       H
f4       ^^tbill  bill
[Hit return to continue]
```

The first column contains the macro name; the second column, the control character sequence used to produce the desired effect; and the final column, the instruction assigned to the macro name. In this example, two key maps are user defined—q and function key F4. The other four are provided with **vi** by default. You can press the New Line key to return control to command mode.

The macro is effective immediately. While **vi** is in command mode, you can issue the **q** command and you will see the command assigned to it displayed briefly at the bottom of the screen. The command is executed and then a message is displayed, reporting the statistics about the file just written.

Setting Up a Function Key Map in Command Mode

An example of mapping a function key to a set of commands follows. You type the following macro definition:

:map #1 i\italic<Ctrl-V><Esc>Ea\roman<Ctrl-V><Esc>↵

What you see on the screen is different from what you typed:

```
:map #1 i\italic^[Ea\roman^[
```

You will see only a single caret (^) from typing Ctrl-V and a single open bracket ([) for typing Esc.

The purpose of this macro is to turn on the italic font for the selected word and then turn on the roman font at the end of the word. A breakdown of each of these commands is given:

`i`	Invokes insert mode.
`\italic`	Requests the italic font.
`<Ctrl-V>`	Represents the Escape character, which must precede the next character.
`<Esc>`	Invokes command mode.
`E`	Moves the cursor to the end of the current word.
`a`	Appends text at the current cursor position.
`\roman`	Requests the roman font.
`<Ctrl-V>`	Represents the Escape character, which must precede the next character.
`<Esc>`	Invokes command mode.

Sample text used in an example of using this macro follows.

```
Book titles are italicized:□Many Miles.
```

Using the F1 key will produce this result:

```
Book titles are italicized:\italicMany Miles\roman.
```

Setting Up a Macro in Input Mode

Two examples of macros you can use while in input mode follow:

:map! z \italic ↵
:map! q \roman ↵

Each time you use either the **z** or **q** macros while **vi** is in input mode, an associated string of text is entered automatically.

If you want to use a literal **z** or **q**, you will have to escape it by preceding it with Ctrl-V.

Sample text used for illustrating the **z** macro follows.

```
Refer to the □
```

Using the **z** macro in input mode causes the mapped text to appear on the screen, which moves the cursor to the position following the new text, shown as follows.

```
Refer to the \italic□
```

Sample text used for illustrating the **q** macro follows.

```
Refer to the \italicUsing the Sprinkler System manual□
```

While in input mode, when you use the **q** macro, the assigned text is inserted at the cursor position, shown as follows.

```
Refer to the \italicUsing the Sprinkler System manual\roman□
```

Setting Up a Function Key Map in Input Mode

An example of mapping a function key to an insert mode function follows:

:map! #2 the Sprinkler System ↵

This macro assigns the function key F2 (the pound sign, **#**, represents a function key), to a string of text.

Sample text used to illustrate this function key map follows.

```
Refer to the Using □
```

In input mode, you use the F2 function key map to insert the mapped text string. The cursor moves to the position following the string, shown as follows.

```
Refer to the Using the Sprinkler System□
```

Assigning Nested Macro Maps

By default, the **remap** editing option is set (refer to Table 2-35), which means that macros can invoke other macros. This means that macros can nest. Several examples of nested maps follow.

```
:set remap
:map ^F w
:map #4 ^F
```

The first mapping, `^F`, performs the **w** command, which advances the cursor to the next word. The second mapping shows that function key, `#4` can also be mapped to the `^F` macro. As long as **remap** is set, `^F` will also perform the **w** command.

To disable nested maps, set the **noremap** editing option. In this case, the first mapping will be enabled. In the example, `^F` will perform the **w** command. Function key `#4` will not; instead it will perform the `^F` command, which pages the screen forward.

You should be careful, however, with the order in which you set up macro definitions. This sequence will not produce the desired results — both key maps will not perform a page forward command.

```
:set remap
:map #4 ^F
:map ^F w
```

Function key `#4` performs `^F`, which pages down a full screen (23 lines). In the next line, `^F`, which is mapped to the **w** command, will overwrite the previous macro containing `Ctrl-F`. Issuing the **map** command in last line mode will show the setup of only the last macro.

As a rule, there should be no matchup between the *definition* field in the first macro and the *macro-name* in a subsequent macro. In this example, the arrangement of the `^F` in these fields will cause problems.

Deleting a Key Map

A macro can be deleted as easily as created, using this command format:

:unmap *macro-name*

If the macro was written for use in input mode, you would use the command **unmap!** instead. You can verify your deletion by issuing the **map** command.

Abbreviation Macro

The abbreviation macro facility is similar to mapping in input mode. The difference between the two is that for the abbreviation macro, the macro name is expanded only if it is entered as a word by itself. If the abbreviation macro name is entered as a part of a word, it will not be recognized as an abbreviation macro so it will not be expanded. The general format used to define a macro follows:

:abbr *macro-name string*

where:

abbr is the keyword representing an abbreviation macro. You can use "**ab**" as a short form of the **abbr** command.

macro-name labels the text string that is invoked each time you issue the macro name.

string is the text to which the macro name expands each time you issue the macro name. An example follows:

:ab mm Missoula, Montana ↵

To ensure that an abbreviation macro has taken effect, you can type the **ab** command in last line mode and the press New Line key.

:ab ↵

```
mm   mm   Missoula, Montana
[Hit return to continue]□
```

The first and second columns give the abbreviation macro name. The final column gives the string to which the macro name expands. You press the New Line key to return **vi** to command mode.

To use this macro, you simply type **mm** preceded and followed by either of these keystrokes that produce whitespace:

Tab
Space
New-line

The abbreviation is expanded immediately to "`Missoula, Montana`" in your file. Sample text used for illustrating the **mm** macro follows.

```
I left my heart in □
```

When you type **mm**, followed by a space, the macro expands and the cursor is positioned immediately following the expanded text, shown as follows.

```
I left my heart in Missoula, Montana□
```

Placement of whitespace around the **mm** macro is crucial. The following sample text illustrates this.

```
I drove through□
```

Using the **mm** macro followed by a space will not cause the macro to be expanded because whitespace does not precede **mm**. The result is shown as follows.

```
I drove throughmm □
```

Deleting Abbreviation Macros

If you no longer need a given abbreviation macro, you can specify the **unab** command followed by the macro name. This command format follows:

unab *macro-name*

An example follows:

:unab mm ↵

The **mm** abbreviation macro is deleted from your list.

Undoing the Effect of a Macro

To undo the effect of a macro, in command mode, use the **u** command. The effect of the macro is undone. To restore the effect of the macro, use the **u** command again. Sample text used to illustrate the **u** command follows.

```
I left my heart in □
```

By typing **mm**, followed by a space, you will see the following result.

```
I left my heart in Missoula, Montana□
```

In the preceding example, you used the **mm** macro to insert the assigned string. The cursor moves to the position following the inserted string. You can undo the effect of the macro expansion by using the **u** command. The text returns to its previous appearance as follows:

```
I left my heart in □
```

To reinstate the text that you just undid, you can use the **u** command again. The result is shown as follows.

```
I left my heart in Missoula, Montana□
```

Miscellaneous

There is a final category of useful commands that do not neatly fit into the primary categories of editing operations. These topics are covered:

- Transposing characters.
- Repeating the last command.
- Joining two lines on one line.
- Clearing the screen and redrawing.
- Changing uppercase to lowercase and vice-versa.
- Getting current editing information.
- Error recovery.
- Alternate ways to invoke **vi**.

Transposing Characters

A quick way to fix transposed characters is to combine the **x** and the **p** commands as **xp**. The command **x** (issued in command mode) deletes the letter; the **p** command places it after the next character.

Notice the error in the next line.

```
What [i]tme is it?
```

By using the **xp** command, you will see the following result.

```
What time is it?
```

The **x** command deletes the "i", causing the text to the right of the cursor to shift left one position. Now the cursor is on the "t", so the **p** command puts back the "i" after the "t".

Repeating the Previous Command

In command mode the **.** (dot) command repeats the effect of the previous command. The dot command (issued in command mode) can be repeated as many times as desired to repeat a command. The dot command is particularly useful when you need to make selective, as opposed to global, changes in a file.

NOTE: Dot commands are ineffective for re-executing commands issued in last line mode. Such a command will repeat the most recent command that altered the buffer.

Sample text used for illustrating the **.** command follows.

```
I wrote 12 memos yesterday.
My dad opened the store in 1912.
I bought 12 pads of paper at the stationery store.
I've asked you 12 times to mail that letter.
```

With the cursor on the `1` in `12`, you issue the **cw** command to change `12` to **a dozen** and press the Escape key to signal an end to the word change. The result is shown as follows.

```
I wrote a dozen memos yesterday.
My dad opened the store in 1912.
I bought 12 pads of paper at the stationery store.
I've asked you 12 times to mail that letter.
```

In the next example, you move the cursor to the next occurrence of `12` that you want to change to **a dozen**. You move the cursor to the third line, skipping over the second line, and position the cursor on the `1` in `12`, shown as follows.

```
I wrote a dozen memos yesterday.
My dad opened the store in 1912.
I bought 12 pads of paper at the stationery store.
I've asked you 12 times to mail that letter.
```

Using the **.** command will repeat the previous command, shown as follows.

```
I wrote a dozen memos yesterday.
My dad opened the store in 1912.
I bought a dozen pads of paper at the stationery store.
I've asked you 12 times to mail that letter.
```

Joining Lines

The **J** command (issued in command mode) joins lines. As an alternative, you can precede the command with a number to specify *n* lines to join, ***n*J**. You can put the cursor anywhere on the first line to be joined and then press the **J** key. The new-line is stripped from the current line, thus allowing the following line to wrap to the preceding line. Remember to use the uppercase **J** because the lowercase **j** moves the cursor down. Sample text to illustrate the **J** command follows.

```
I wrote 12 memos yesterday.
My dad opened the store in 1912.
I bought 12 pads of paper at the stationery store.
I've asked you 12 times to mail that letter.
```

Using the **J** command produces the following result.

```
I wrote 12 memos yesterday.□ My dad opened the store in 1912.
I bought 12 pads of paper at the stationery store.
I've asked you 12 times to mail that letter.
```

The amount of text to be wrapped to the preceding line will depend on the screen length, which you can set through the wrapmargin option (refer to "Setting Up Your vi Environment" in this chapter for more information on these methods.)

Text is wrapped automatically up to the current line. Notice that two spaces are automatically inserted after the final punctuation (.) following "`afternoon.`" If there had been no punctuation, one space would be inserted between the two joined text objects. This is done to maintain proper spacing both inside a sentence and between sentences. If you want continuously wrapping text with no new lines, you can press the **J** key repeatedly.

Clearing and Refreshing the Terminal Window

If your terminal screen contains spurious marks (such as from the result of broadcast messages or "noise" on a modem connection), you can restore the current window to its proper image using the Ctrl-L command in command mode.

Changing Cases

A quick way to change any lowercase letter to uppercase, or vice-versa, is by putting the cursor on the letter to be changed and, in command mode, typing a ~ (tilde). For example, to change the letter **a** to A, with the cursor on **a**, use the ~ command. This command also causes the cursor to move one position forward. To convert an entire line (or part of a line) of text in one case to the alternate case, you can use the ~ command repeatedly. As the cursor moves forward, characters are converted to the alternate case. Numbers, punctuation, and blanks are unaffected. Sample text follows.

```
The Dow is down.
```

When you use use the ~ command 16 times, you will see this result.

```
tHE dOW IS DOWN.
```

Each time you use the ~ command, the current character is changed to the alternate case.

This command can also be modified by a *number*. Using the **16~** command will produce the same result.

Getting Current Editing Information

While you are editing, **vi** is keeping track of some information about your text. You can get a report of this information at any time during a **vi** editing session while in command mode. Sample text follows.

```
If you plan to be in the vicinity □
of the Central City Coliseum tonight
```

Using the Ctrl-G command will produce this result:

```
"cab.alert" [modified] line 36 of 116 --34%--
```

where:

"`cab.alert`" is the filename.

`[modified]` means that the file has been changed since it was last saved.

`line 36` is the current line number.

`of 116` is the total number of lines in file.

`--34%--` is the percentage of total lines represented by first line through current line.

Error Recovery

There are two methods for recovering from your errors or computer errors.

- Undoing the last command.
- Recovering a file lost due to a system crash.

Undoing the Previous Command

Vi allows you to undo the last command you issued in command mode. This is extremely helpful in allowing you to restore the file to its original state following an undesired or mistaken command. Table 2-41 lists the undo commands.

Table 2-41 Undo Commands

Command	Definition
u	Undoes the last command. The cursor does not have to be on the line where the last command was executed. The **u** command will undo the effect of the **U** command (see next table entry).
U	Undoes all edits on a single line as long as the cursor remains on that line. The **U** command has no effect if you move the cursor from that line. The **U** command will undo the effect of the **u** command.
:undo	In last line mode, undoes the previous command. Issuing this command a second time reverses the **undo** command in an identical manner to multiple uses of the **u** command.

If you delete several lines by mistake, for example, you can restore them with the **u** command. In fact, you can change your mind and delete those lines again with the **u** command. With **u**, you can go back and forth — it's a toggle.

As an alternative, after you have done a number of editing changes on a line, you can fix them all and restore the line to its original state with the **U** command. This command backs out all changes made since the cursor was positioned on the current line. Sample text for illustrating the **u** and **U** commands follows.

$\boxed{\texttt{D}}$`ogs are man's best friend.`

When you use the **cw** to change `dogs` to **cats**, and then press the Escape key, you will see the following result.

`Cat`$\boxed{\texttt{s}}$` are man's best friend.`

To undo this change, you would use the **u** command. The result is shown as follows.

$\boxed{\texttt{D}}$`ogs are man's best friend.`

To reinstate the original change, you would use the **u** command again. The result is shown as follows.

`Cat`$\boxed{\texttt{s}}$` are man's best friend.`

Using the **U** command will undo the effect of the **u** command. The result is shown as follows.

$\boxed{\texttt{D}}$`ogs are man's best friend.`

Recovering Lost Files

If your computer interrupts or disconnects (crashes), **vi** will be terminated without warning you, and worse yet, without writing your file to disk. The system does store an image of your file, which you can recover and continue to work on. You can lose as much as the last 12 lines or parts of lines changed. Any marks or text saved in registers will be lost. The following steps tell how to recover a file.

Recovering a File from the Shell

1) Log in to your computer system.

2) Go to the directory in which the lost file was located.

3) Invoke **vi** from the shell using the following command format:

   vi −r *filename*

   where:

   the **−r** option means to recover.

   filename names the file being edited during the crash.

You could lose as much as the last 12 lines or parts of lines changed.

NOTE: You will normally receive electronic mail when you log in to the system again, after your computer has been restored, giving you the name of the file that was saved for you. You can invoke **vi** as **vi −r** to get a list of these files saved for you, and to find out their exact names.

Trouble Saving File in a vi Session

There may be instances when you try to save a file and you get a message from the system informing you that the file system is full or unavailable. Follow these steps to preserve a copy of your buffer:

1) From last line mode, issue this command:

   :preserve ↵.

   This command saves the current buffer in a reserved directory on the system.

2) In the meantime, you can continue your work on other files as normal. After some time has passed, you can check to see if any system resources have been freed by issuing this command from **vi** last line mode:

   :recover *filename* ↵

   If you can successfully recover your file, try writing it again to a permanent file. If system resources are still unavailable, keep trying until there are adequate resources or try the following step.

3) Type this command from last line mode:

   :!df ↵

   This commands checks the availability of resources on other file systems. If there are resources on a system to which you have write permissions, write your file using this command:

   : w! *pathname* ↵

   Continue your work at that location and then move the file back to your file system when resources are available.

End of Chapter

Chapter 3
Using the Line Editor: ed

This chapter discusses the line editor, **ed**. **Ed** is versatile and requires little computer time to perform editing tasks. It can be used on any type of terminal. The examples of command lines and system responses in this chapter will apply to your terminal, whether it is a video display terminal or a hardcopy terminal. The **ed** commands can be typed in at your terminal or they can be used in a shell program. See *Using the DG/UX™ System* for more information on using **ed** in a shell program.

During an editing session with **ed**, you are altering the contents of a file in a temporary (or working) buffer, where you work until you have finished creating or changing your text. When you edit an existing file, a copy of that file is placed in the buffer and your changes are made to this copy. The changes have no effect on the original file until you instruct **ed** with the write command to move the contents of the buffer into the file.

Ed always points to a single line in the buffer called the current line. When you invoke **ed** to edit an existing file, **ed** makes the file's last line the current line so you can start appending text easily. Unless you specify the number of a different line or range of lines, **ed** will perform each command you issue on the current line. In addition to letting you change, delete, or add text on one or more lines, **ed** allows access to files other than the one you are editing — you can read in the contents of another file or overwrite another file's contents with lines from the buffer.

After you have read through this chapter and tried the examples, you will have a good working knowledge of **ed**. The chapter covers all you need to know to use **ed**, including the following topics:

- Invoking **ed**.
- Displaying text.
- Entering text.
- Deleting text.
- Searching for text and performing global changes to text.
- Rearranging text.

- Reading text from and writing text to a file.
- Saving text and quitting **ed**.

The notation conventions used in this chapter are described in the Preface of this manual.

Most of the examples in this chapter refer to a file called **try-me**. The steps for creating the initial version of this file follows.

```
$ cat > try-me ↵
This is the first line of text. ↵
This is the second line, ↵
and this is the third line. ↵
This is the fourth line. ↵
<Ctrl-D>
```

You display the file's contents as follows.

```
$ cat try-me ↵
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line.
$
```

As the chapter progresses, the various examples may change or add to the text in file **try-me**.

Invoking ed

Use this format to invoke **ed** from the shell:

ed [*options*]*filename*

where:

options are available for customizing the behavior of **ed**. See the **ed**(1) man page in the *User's Reference for the DG/UX™ System* for more information.

The following example shows how to invoke **ed** to edit the file **try-me**:

```
$ ed try-me ↵
```

If you are editing a new file, **ed** responds with a question mark (?) and the filename. If you are editing an existing file, **ed** returns the number of characters in the file.

Once you have invoked **ed**, you are in command mode and can use any of the **ed** commands. Table 3-1 summarizes the basic **ed** commands.

Table 3-1 Summary of ed Commands

Command	Definition
↵	Displays the next line in the buffer.
+	Displays the next line in the buffer.
–	Displays the previous line in the buffer.
p	Prints addressed lines on your screen.
n	Prints addressed lines with line numbers on your screen.
a	Changes control to input mode, appending typed text after the current line. Terminate input mode by typing a period alone on a line and pressing the New Line key.
i	Inserts text before the addressed line in the buffer.
c	Changes the text on the addressed line(s) to new text.
.	Quits input mode, returning **ed** to command mode.
d	Deletes text on addressed lines.
u	Undoes previous command, restoring any text that was deleted or changed.
s	Substitutes the first occurrence of a string with a replacement string.
g	Globally substitutes each instance of a string within a specified range with a replacement string.
m	Moves the addressed line(s) of text (deleting the original) to another buffer location.
t	Copies (duplicates) the addressed lines to another buffer location.
j	Joins the current line with the following line.
r	Reads the contents of a file into the current buffer.
w	Writes the addressed lines of the buffer or the entire buffer to a file. If the file already exists, its contents will be overwritten.
H	Turns on help message mode so you receive help messages when **ed** detects an error. Usually **ed** returns a question mark (?) alone on a line to report an error.
P	Turns prompting mode on and off. In prompting mode, **ed** prompts you after completing a command.
q	Quits **ed**, without writing buffer contents to the file, and returns to the shell.

The following section describes the general format that applies to **ed** commands, and the sections following describe the various **ed** commands in detail.

General Format of ed Commands

Ed commands have a simple and regular format:

[*address1* [*,address2*]] *command* [*argument*]

where:

address1,address2 are the positions of lines in the buffer. *Address1* through *address2* specifies a range of lines that will be affected by the *command*. If *address2* is omitted, the command will affect only the line specified by *address1.*

command is one character and tells the editor what task to perform.

argument is a filename, another line address, or some indicator telling what parts of the text will be modified.

This format will become clearer when you begin to experiment with the **ed** commands.

Line Addressing

A line address is a character or group of characters that identifies a line of text. Before **ed** can execute commands that add, delete, move, or change text, it must know the line address of the affected text. The line address precedes the command:

[*address1* [*,address2*]] *command*

Both *address1* and *address2* are optional. Specify *address1* alone to request action on a single line of text; specify both *address1* and *address2* to request a range of lines. If you do not specify any *address*, **ed** assumes that the line address is the current line. Table 3-2 shows the symbols and commands you can use for addressing lines.

Table 3-2 Summary of Line Address Syntax

Address	Description
n	Represents line number *n* in the buffer.
.	Represents the current line (the line most recently acted on by an **ed** command).
$	Represents the last line in the buffer.
,	Represents all lines in the buffer (the set of lines from line 1 through the last line).
;	Represents the set of lines from the current line through the last line.
+ *n*	Represents the line that is located *n* lines after the current line.
− *n*	Represents the line that is located *n* lines before the current line.
/*abc*	Represents the next line in the buffer (searching from the current line forward) that matches the pattern *abc*.
?*abc*	Represents the next line in the buffer (searching from the current line backward) that matches the pattern *abc*.
g/*abc*	Represents the set of all lines that contain the pattern *abc*.
v/*abc*	Represents the set of all lines that do *not* contain the pattern *abc*.

To find out the absolute line number of a particular line, issue the **=** command, preceding it with an address indicating that line. The following example shows how to get the line number of the last line in a file, in this case, a file four lines long. Issuing this command does not change the current line.

```
$= ↵
4
```

Numerical Addresses

Ed gives a *numerical address* to each line in the buffer. The first line of the buffer is 1, the second line is 2, and so on, for each line in the buffer. Any line can be accessed by **ed** with its line address number. To see how line numbers address a line, enter **ed** with the file **try-me** and type a number.

```
$ ed try-me ↵
110
1 ↵
This is the first line of text.
3 ↵
and this is the third line.
```

When you issue an address alone as a command, **ed** assumes the print (**p**) command. If the address indicates only one line, **ed** prints the line and makes it the current line. If the address indicates more than one line, **ed** prints only the last line and makes it the current line. To print all of the lines implied by an address, issue the **p** command explicitly with the address.

Numerical line addresses frequently change in the course of an editing session. Later in this chapter you will create lines, delete lines, or move a line to a different position. This will change the line address numbers of some lines. The number of a specific line is always the current position of that line in the editing buffer. For example, if you add five lines of text between lines 5 and 6, line 6 becomes line 11. If you delete line 5, line 6 becomes line 5.

Symbolic Addresses

Ed offers a number of symbolic addresses that you can use to refer to a line or set of lines. These symbols do not require that you know the numbers of any lines in the buffer.

Current Line

The current line is the line most recently acted on by any **ed** command. If you have just entered **ed** with an existing file, the current line is the last line of the buffer. The symbol for the address of the current line is a period. Therefore you can display the current line simply by typing a period (**.**) and pressing the New Line key.

Try this command in the file **try-me**:

```
$ ed try-me ↵
110
. ↵
This is the fourth line.
```

The **.** is the address. Because a command is not specified after the period, **ed** executes the default command **p** and displays the line found at this address.

To get the line number of the current line, type the following command:

```
.= ↵
```

Ed responds with the line number. For example, in the **try-me** file, the current line is 4.

```
. ↵
This is the fourth line.
.= ↵
4
```

Last Line

The symbolic address for the last line of the buffer is **$**. To verify that **$** accesses the last line, access the **try-me** file with **ed** and specify this address on a line by itself. Keep in mind that when you first invoke **ed** on a file, your current line is the last line of the file.

```
$ ed try-me ↵
110
.↵
This is the fourth line.
$↵
This is the fourth line.
```

NOTE: Remember that the **$** address within **ed** is not the same as the $ prompt from the shell.

All Lines

When used as an address, a comma (**,**) refers to all the lines in the buffer, from the first through the last line. It is an abbreviated form of the string **1,$**. Try this shortcut to print the contents of **try-me**:

```
,p ↵
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line.
```

Current Line through the Last Line

The semicolon (**;**) represents a set of lines beginning with the current line and ending with the last line in the buffer. It is equivalent to the symbolic address **.,$**. Try it with the file **try-me**:

```
2p ↵
This is the second line,
;p ↵
This is the second line,
and this is the third line.
This is the fourth line.
```

Relative Addresses

You may often want to address lines with respect to the current line. You can do this by adding or subtracting a number of lines from the current line with a plus (**+**) or a minus (**−**) sign. Addresses derived in this way are called addresses. To experiment with relative line addresses, add several more lines to your file **try-me**, as shown in the following screen. Also, write the buffer contents to the file so your additions will be saved:

```
$ ed try-me ↵
110
. ↵
This is the fourth line.
a ↵
five ↵
six ↵
seven ↵
eight ↵
nine ↵
ten ↵
. ↵
w ↵
140
```

Now try adding and subtracting line numbers from the current line.

```
4 ↵
This is the fourth line.
+3 ↵
seven
−5 ↵
This is the second line,
```

The following example shows what happens if you ask for a line address that is greater than the last line or if you try to subtract a number greater than the current line number.

```
5 ↵
five
−6 ↵
?
.= ↵
5
+7 ↵
?
```

Notice that the current line remains at line 5 of the buffer. The current line changes only if you give **ed** a correct address. The **?** response means there is an error. The section "Other Useful Commands and Information," at the end of this chapter, explains how to get a help message that describes the error.

Character String Addresses

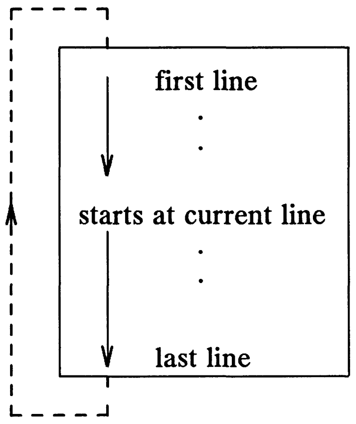
You can search forward or backward in the buffer for a line containing a particular character string. To do so, precede the string or pattern with the appropriate delimiter.

Delimiters mark the boundaries of character strings; they tell **ed** where a string starts and ends. The most common delimiter is **/** (slash), used in the following format:

/pattern

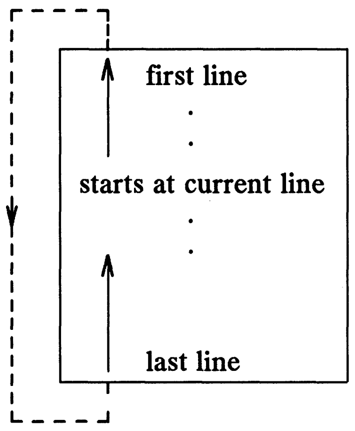
When you specify a pattern preceded by **/**, **ed** begins at the current line and searches forward (down through subsequent lines in the buffer) for the next line containing *pattern*. When the search reaches the last line of the buffer, **ed** wraps around to the beginning of the buffer and continues its search from line 1. If **ed** does not find the pattern, the search ends at the current line.

The following rectangle represents the editing buffer. The path of the arrows shows the search initiated by **/**:



Another useful delimiter is **?**. If you specify a pattern preceded by **?**, (such as **?***pattern*), **ed** begins at the current line and searches backward (up through previous lines in the buffer) for the next line containing the *pattern*. If the search reaches the first line of the buffer, it will wrap around and continue searching upward from the last line of the buffer. Again, if **ed** does not find the pattern, the search ends at the current line.

The following rectangle represents the editing buffer. The path of the arrows shows the search initiated by **?**:



Experiment with these kinds of searches on the file **try-me**.

```
$ ed try-me ↵
140
. ↵
ten
?first ↵
This is the first line of text.
/fourth ↵
This is the fourth line.
/junk ↵
?
```

In this example, **ed** found the specified strings **first** and **fourth**. Then, because no command was given with the address, it executed the **p** command by default, displaying the lines it had found. When **ed** cannot find a specified string (such as **junk**), it responds with a `?`.

When you issue **/** without specifying a search pattern, **ed** assumes you want to search for the last pattern specified. In the file **try-me**, for example, you can search for the pattern **line** by issuing **/line** once, then search for further occurrences of **line** merely by issuing **/** alone:

```
. ↵
This is the first line of text.
/line ↵
This is the second line,
/ ↵
and this is the third line.
/ ↵
This is the fourth line.
/ ↵
This is the first line of text.
```

Notice that after **ed** has found all occurrences of the pattern between the line where you requested a search and the end of the buffer, it wraps around to the beginning of the buffer and continues searching.

Specifying a Range of Lines

There are two ways to request a group of lines. You can specify a range of lines, such as *address1* through *address2*, or you can specify a global search for all lines containing a specified pattern.

The simplest way to specify a range of lines is to use the line numbers of the first and last lines of the range, separated by a comma (**,**). Place this address before the command. For example, if you want to display lines 2 through 7 of the editing buffer, give *address1* as 2 and *address2* as 7 in the following format:

```
2,7p ↵
```

Try this on the file **try-me**:

```
2,7p ↵
This is the second line,
and this is the third line.
This is the fourth line.
five
six
seven
```

If you type **2,7** without the **p** command, **ed** prints only line **7**, the last line of the range of addresses.

Relative line addresses can also be used to request a range of lines. Be sure that *address1* precedes *address2* in the buffer. Relative addresses are calculated from the current line, as the following example shows:

```
4 ↵
This is the fourth line
–2,+3p ↵
This is the second line,
and this is the third line.
This is the fourth line.
five
six
seven
```

Global Searches

There are four commands you can use for global searches and other operations. Table 3-3 shows these commands.

Table 3-3 Summary of Global Search Commands

Command	Definition
g	Searches for lines matching a given pattern and performs a command on those lines.
v	Searches for lines that do *not* match a given pattern and performs a command on those lines.
G	Searches for lines matching a given pattern and, for each line found, lets you issue a command.
V	Searches for lines not matching a given pattern and, for each line found, lets you issue a command.

The commands follow these formats:

[*address* [*,address*]] **g/***pattern***/***command*
[*address* [*,address*]] **v/***pattern***/***command*
[*address* [*,address*]] **G/***pattern*
[*address* [*,address*]] **V/***pattern*

where:

address1 [*,address2*] is the range of lines that will be searched. The search can be on one line (*address1*) or on a range of lines (*address1* through *address2*). If you provide no address, **ed** searches the entire buffer.

g is the global search command.

pattern is the regular expression search pattern that defines which of the addressed lines the command will affect.

command is the command to be executed for the matching lines.

If you do not specify a command for the **g** and **v** commands, **ed** assumes **p** as the default and prints the lines found.

You cannot include a command on the **G** command line. You provide only the search pattern expression. Internally, **ed** performs the entire global search first, marking the lines that satisfy the search pattern. After marking the lines, **ed** prompts you with the first one found, printing it on your screen and making it the current line. You may then issue a command or simply press New Line. **Ed** executes the command, if you provided one, and continues to the next marked line. To repeat the last command issued within the current invocation of **G**, issue **&** alone on the command line. You cannot issue the **a**, **c**, **i**, **g**, **G**, **v**, or **V** command while the **G** command is working. You can halt **G** prematurely by pressing the Delete key or the Break key.

The **V** command operates the same as **G** except that, instead of selecting lines that match the given pattern, **V** selects lines that do not match the pattern. The same options and restrictions apply to **V** as apply to **G**.

The following example shows how you could use the **g** command to print all lines in the **try-me** file that contain the word "`line`."

```
g/line/p ↵
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line.
```

The following example shows how you could use the **v** command to print all lines not containing the word "`line`."

```
v/line/p ↵
five
six
seven
eight
nine
ten
```

The following example shows how you could use the **G** command to find all lines containing the word "`line`" and then display them one by one, allowing you to execute a command after each one. If you were to issue the **d** command after being prompted by the first line of matching text, you would delete that line, as shown. Pressing New Line leaves the second line untouched, and issuing the **d** command again for the third and fourth lines deletes them both.

```
G/line ↵
This is the first line of text.
d ↵
This is the second line,
↵
and this is the third line.
d ↵
This is the fourth line.
d ↵
```

The following example shows a command that would match all lines not containing the word **line**. Pressing New Line after the first matching line, `five`, leaves it untouched, while pressing **n** for the second matching line causes **ed** to print it with its line number before continuing the search. The **d** command deletes the next line, and pressing the Delete key after the eighth line interrupts the command.

```
V/line ↵
five
↵
six
n ↵
6       six
seven
d ↵
eight
<Del>
```

The various **ed** commands all follow the basic format described in this section. Where the commands accept line addresses, they accept any of the address types described here. The following sections describe the **ed** commands in detail.

Displaying Text

Ed provides two commands for displaying lines of text in the editing buffer: **p** and **n**. Table 3-4 summarizes these commands.

Table 3-4 Summary of Commands for Displaying Text

Command	Definition
p	Displays specified lines of text in the editing buffer.
n	Displays specified lines of text in the editing buffer with their numerical line addresses.

Displaying Text Alone (p)

You have already used the **p** command in several examples. You are probably now familiar with its general format:

[*address1* [,*address2*]] **p**

The **p** command does not take arguments. However, it can be combined with a substitution command line. This will be discussed later in this chapter.

Table 3-5 shows example line addresses and their effect on the **p** command.

Table 3-5 Sample Addresses for Displaying Text

Example	Effect
1,$p ↵	Displays all lines in the buffer.
–5p ↵	Moves backward five lines from the current line and displays the line found there.
+2p ↵	Moves forward two lines from the current line and displays the line found there.
1,/x/p ↵	Displays the set of lines from line 1 through the first line after the current line that contains the character **x**. It is important to enclose the letter **x** between slashes so that **ed** can distinguish between the search pattern (**x**) and the (**p**) command.

Displaying Text with Line Addresses (n)

The **n** command displays text and precedes each line with its numerical line address. It is helpful when you are deleting, creating, or changing lines. The general command line format for **n** is the same as that for **p**.

[*address1* [*,address2*]] **n**

Like **p**, **n** does not take arguments, but it can be combined with the substitute command. Try using **n** in the **try-me** file:

```
$ ed try-me ↵
140
1,$n ↵
1       This is the first line of text.
2       This is the second line,
3       and this is the third line.
4       This is the fourth line.
5       five
6       six
7       seven
8       eight
9       nine
10      ten
```

Entering Text

Ed has three basic commands for entering new text. Table 11-6 summarizes these commands.

Table 3-6 Summary of Commands for Creating Text

Command	Definition
a	Appends text after the specified line in the buffer.
i	Inserts text before the specified line in the buffer.
c	Changes the text on the specified line(s) to new text.
.	Quits text input mode and returns **ed** to command mode.

Appending Text (a)

The append command **a** allows you to add text *after* the current line or a specified address in the buffer. The general format for the append command line is:

[*address1*] **a**

Specifying an address is optional. The default value of *address1* is the current line.

In the following example, a new file called **new-file** is created. In the first append command line, the default address is the current line. In the second append command line, line 1 is specified as *address1*. The lines are displayed with **n** so that you can see their numerical line addresses. Remember, the append mode is ended by typing a period (**.**) on a line by itself.

```
$ ed new—file ↵
?new-file
a ↵
Create some lines ↵
of text in ↵
this file. ↵
. ↵
1,$n ↵
1       Create some lines
2       of text in
3       this file.
1a ↵
This will be line 2 ↵
This will be line 3 ↵
. ↵
1,$n ↵
1       Create some lines
2       This will be line 2
3       This will be line 3
```

```
4       of text in
5       this file.
```

Notice that after you append the two new lines, the line, "`of text in`" (originally line 2), becomes line 4.

You can take shortcuts to places in the buffer where you want to append text by combining the append command with symbolic addresses. The following three command lines allow you to move through and add to the text quickly in this way.

a ↵	Appends text after the current line.
$a ↵	Appends text after the last line in the buffer.
0a ↵	Appends text before the first line in the buffer (at a symbolic address called line 0).

To try using these addresses, create a one-line file called **lines** and type the examples shown in the following screens.

```
a ↵
This is the current line. ↵
. ↵
p ↵
This is the current line.
.a ↵
This line is after the current line. ↵
. ↵
–1,.p ↵
This is the current line.
This line is after the current line.
```

```
$a ↵
This is the last line now. ↵
. ↵
$ ↵
This is the last line now.
```

```
0a ↵
This is the first line now. ↵
This is the second line now. ↵
The line numbers change ↵
as lines are added. ↵
. ↵
1,4n ↵
1       This is the first line now.
2       This is the second line now.
3       The line numbers change
4       as lines are added.
```

Because the append command creates text after a specified address, the last example refers to the line before line 1 as the line after line 0. To avoid such circuitous references, use another command provided by the editor: the insert command **i**.

Inserting Text (i)

The insert command **i** allows you to add text before a specified line in the editing buffer. The general command line format for **i** is the same as that for **a**:

[*address1*] **i**

As with the append command, you can insert one or more lines of text. To quit input mode, you must type a period (**.**) alone on a line.

Create a file called **insert** in which you can try the insert command **i**:

```
$ ed insert ↵
?insert
a ↵
Line 1 ↵
Line 2 ↵
Line 3 ↵
Line 4 ↵
. ↵
w ↵
28
```

Now insert one line of text above line 2 and another above line 1. Use the **n** command to display all the lines in the buffer:

```
2i ↵
This is the new line 2. ↵
. ↵
1,$n ↵
1       Line 1
2       This is the new line 2.
3       Line 2
4       Line 3
5       Line 4
1i ↵
This is the beginning. ↵
. ↵
1,$n ↵
1       This is the beginning
2       Line 1
3       Now this is line 2
4       Line 2
5       Line 3
6       Line 4
```

Experiment with the insert command by combining it with symbolic line addresses, as follows:

```
i ↵
$i ↵
```

Changing Text (c)

The change command **c** erases all specified lines and allows you to create one or more lines of text in their place. Because **c** can erase a range of lines, the general format for the command line includes two addresses.

[*address1* [*,address2*]] **c**

where:

address1 is the first and *address2* is the last of the range of lines to be replaced by new text. To erase one line of text, specify only *address1*. If no address is specified, **ed** assumes the current line is the line to be changed.

The change command puts you in text input mode. To leave input mode, type a period alone on a line.

Now create a file called **change** in which you can try this command. After entering the text shown in the screen, change lines 1 through 4 by typing **1,4c**:

```
1,5n ↵
1       line 1
2       line 2
3       line 3
4       line 4
5       line 5
1,4c ↵
Change line 1 ↵
and lines 2 through 4 ↵
. ↵
1,$n ↵
1       change line 1
2       and lines 2 through 4
3       line 5
```

Now experiment with **c** and try to change the current line:

```
. ↵
line 5
c ↵
This is the new line 5.
. ↵
. ↵
This is the new line 5.
```

Deleting Text

This section discusses two ways you can delete text in **ed**. One way works when you are in command mode: the **d** command deletes a line and **u** undoes the last command. The other way, which you can use while in text input mode, involves the line discipline you use to delete a character or kill a line in the shell. For more information on shell line discipline, see *Using the DG/UX™ System*.

Table 3-7 summarizes the **ed** commands and shell commands you can use to delete text.

Table 3-7 Summary of Commands for Deleting Text

Command	Definition
In Command mode:	
d	Deletes one or more lines of text.
u	Undoes the previous command.
Ctrl-U	Deletes the current command line.
In Text Input mode (shell defaults):	
Ctrl-U	Deletes the current line.
Del	Deletes the last character typed.

Deleting Lines (d)

You delete lines of text with the delete command **d**. The general format for the **d** command line is:

[*address1* [,*address2*]] **d**

where:

address1 and *address2* specify the range of lines you want to delete. A single address deletes only that line. Specifying no address deletes the current line.

The next example displays lines 1 through 5 and then deletes lines 2 through 4.

```
1,5n ↵
1       1 horse
2       2 chickens
3       3 ham tacos
4       4 cans of mustard
```

```
5       5 bails of hay
2,4d ↵
1,$n ↵
1       1 horse
2       5 bails of hay
```

You can delete the last line in the buffer like this:

```
$d ↵
```

One of the most common errors in **ed** is forgetting to type a period to leave text input mode. If you do this, you may add unwanted text to the buffer. The following example shows how you could accidentally add a print command (**1,$p**) to the text before leaving input mode. Because this line is the last one in the text, it becomes the current line. You can use **d** to delete it.

```
$a ↵
Last line of text ↵
1,$p ↵
. ↵
p ↵
1,$p
.d ↵
p ↵
Last line of text.
```

Before experimenting with the delete command, you may first want to learn about the undo command **u**.

Undoing the Previous Command (u)

The command **u** (short for undo) nullifies the last command and restores any text changed or deleted by that command. It takes no addresses or arguments. The format is:

```
u
```

One purpose for which the **u** command is useful is to restore text you have mistakenly deleted. If you delete all the lines in the buffer and then type **p**, **ed** will respond with a `?` since there are no more lines in the buffer. Use the **u** command to restore them.

```
1,$p ↵
This is the first line.
This is the middle line.
This is the last line.
1,$d ↵
p ↵
?
u ↵
p ↵
This is the last line.
```

Now experiment with **u**: use it to undo the append command.

```
. ↵
This is the only line of text
a ↵
Add this line ↵
. ↵
1,$p ↵
This is the only line of text
Add this line
u ↵
1,$p ↵
This is the only line of text
```

NOTE: You cannot use **u** to undo the write command **w** or the quit command **q**. You can, however, use **u** to undo an undo command. The effect of undoing an undo command is the same as not performing any undo command at all: if you issue two undo commands one after the other, the second effectively cancels the first.

Deleting in Text Input Mode (Ctrl-U and Del)

While in text input mode, you can correct the current line of input with the same keys you use to correct a shell command line. Normally, Ctrl-U performs the line kill function while the Delete key performs the character delete function. A line kill deletes the entire current line, and a character delete deletes the preceding character.

These key functions are part of the line discipline you set in the shell. You can reassign the line kill and character delete functions to other keys if you prefer. For more information on changing line discipline, see *Using the DG/UX™ System*.

Substituting Text (s)

You can change your text with a substitute command. This command replaces the first occurrence of a string of characters with new text. The general command line format follows.

[*address1* [*,address2*]] **s***/old-text/new-text* [*/command*]

where:

address1 [*,address2]* defines the range of lines being addressed by **s**. The address can be one line, (*address1*), a range of lines (*address1* through *address2*), or a global search address. If no address is given, **ed** makes the substitution on the current line.

s is the substitute command.

/old-text is the text to be replaced. This argument is usually delimited by slashes, but can be delimited by other characters such as a question mark ? or a period. The command replaces the first occurrence of these characters that it finds in the text.

/new-text is the text that will replace the old text. This argument is also delimited by slashes or the same delimiters used to specify *old-text*.

/command is any one of the following four commands:

g Change all occurrences of *old-text* on the specified lines.

l Display the last line of substituted text, including nonprinting characters. See the last section of this chapter, "Other Useful Commands and Information."

n Display the last line of the substituted text with its numerical line address.

p Display the last line of substituted text.

Current Line

The simplest example of the substitute command is making a change to the current line. You do not need to give a line address for the current line.

s/*old-text***/***new-text***/**

The next example contains a typing error. While the line that contains it is still the current line, you make a substitution to correct it. The old text is the **ai** of **airor** and the new text is **er**.

```
a ↵
In the beginning, I made an airor.
. ↵
.p ↵
In the beginning, I made an airor.
s/ai/er/ ↵
```

Notice that **ed** gives no response to the substitute command. To verify that the command has succeeded in this case, you either have to display the line with **p** or **n**, or include **p** or **n** as part of the substitute command line. In the following example, **n** is used to verify that the word **file** has been substituted for the word **toad**.

```
.p ↵
This is a test toad
s/toad/file/n ↵
1       This is a test file
```

Ed allows you this shortcut: it prints the results of the command automatically if you omit the last delimiter after the *new-text* argument:

```
.p ↵
This is a test file
s/file/frog ↵
This is a test frog
```

Single Line Other than the Current Line

To substitute text on a line that is not the current line, include an address in the command line, as follows:

[*address1*] **s/***old-text***/***new-text***/**

For example, in the following screen the command line includes an address for the line to be changed (line 1) because the current line is line 3:

```
1,3p ↵
This is a pest toad
testing testing
come in toad
```

```
. ↵
come in toad
1s/pest/test ↵
This is a test toad
```

The preceding example shows how omitting the last delimiter causes **ed** to print the new line automatically after the operation.

Range of Lines

You can make a substitution on a range of lines by specifying the first address (*address1*) through the last address (*address2*).

[*address1* [*,address2*]] **s/***old-text***/***new-text***/**

If **ed** does not find the pattern to be replaced on a line, no changes are made to that line.

In the following example, all the lines in the buffer are addressed for the substitute command. However, only the lines that contain the string **es** (the *old-text* argument) are changed.

```
1,$p ↵
This is a test toad
testing testing
come in toad
testing 1, 2, 3
1,$s/es/ES/n ↵
4       tESting 1, 2, 3
```

When you specify a range of lines and include **p** or **n** at the end of the substitute line, only the last line changed is printed.

To display all the lines in which text was changed, use the **n** or **p** command with the address **1,$**.

```
1,$n ↵
1       This is a tESt toad
2       tESting testing
3       come in toad
4       tESting 1, 2, 3
```

Notice that only the first occurrence of **es** (on line 2) has been changed. To change every occurrence of a pattern, use the **g** command, described in the next section.

Global Substitutions (g)

One of the most versatile tools in **ed** is global substitution. By placing the **g** command after the last delimiter on the substitute command line, you can change every occurrence of a pattern on the specified lines. Try changing every occurrence of the string **es** in the last example. If you are following along, doing the examples as you read this, remember you can use **u** to undo the last substitute command.

```
u ↵
1,$p ↵
This is a test toad
testing, testing
come in toad
testing 1, 2, 3
1,$s/es/ES/g ↵
1,$p ↵
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Another method is to use a global search pattern as an address instead of the range of lines specified by **1,$**.

```
1,$p ↵
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/test/s/es/ES/g ↵
1,$p ↵
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

If the global search pattern is unique and matches the argument *old-text* (text to be replaced), you can use an **ed** shortcut: specify the pattern once as the global search address and do not repeat it as an *old-text* argument. **Ed** will remember the pattern from the search address and use it again as the pattern to be replaced.

g/*old-text***/s//***new-text***/g**

NOTE: When you use this shortcut, be sure to include two slashes (//) after the **s**.

```
1,$p ↵
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/es/s//ES/g ↵
1,$p ↵
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Experiment with other search pattern addresses:

/pattern
?pattern
v*/pattern*

You can combine any of these kinds of searches with a substitution command. In the following example, the **v***/pattern* search format is used to locate lines that do not contain the pattern "`testing`." Then the substitute command **s** is used to replace the existing pattern **in** with a new pattern **out** on those lines.

```
v/testing/s/in/out ↵
This is a test toad
come out toad
```

Notice that the line "`This is a test toad`" was also printed, even though no substitution was made on it. When the last delimiter is omitted, all lines found with the search address are printed, regardless of whether or not substitutions have been made on them.

Now search for lines that do contain the pattern "`testing`" with the **g** command.

```
g/testing/s//jumping ↵
jumping testing
jumping 1, 2, 3
```

Notice that this command makes substitutions only for the first occurrence of the pattern "`testing`" in each line. Once again, the lines are displayed on your terminal because the last delimiter has been omitted.

Metacharacters for Pattern Matching

There are a number of special characters, called metacharacters, that have special meanings when they appear in search patterns. Consider what happens when you try to substitute a $, as in the following example:

```
.↵
I lost my $ in Las Vegas.
s/$/money ↵
I lost my $ in Las Vegas.money
```

You will find that the replacement text appears at the end of the line rather than in place of the $. The reason is that the $ is a metacharacter representing the end of the line. There are also a number of other metacharacters, and when you use one or more of them in a search pattern, the pattern constitutes a regular expression.

Regular expressions provide a shorthand for expressing search or substitution patterns. For information on forming regular expressions, see Appendix A.

In addition to the standard metacharacters, **ed** adds two special characters of its own. Table 3-8 summarizes the metacharacters and other special characters you can use in search and substitution patterns.

Table 3-8 Summary of Pattern-matching Characters

Character(s)	Function
.	Matches any one character.
*	Matches zero or more occurrences of the preceding character.
.*	Matches zero or more occurrences of any characters.
^	Matches the beginning of the line.
$	Matches the end of the line.
\	Escapes the following character so **ed** does not interpret it as a special character.
[...]	Matches the first occurrence of a character in the brackets.
[^...]	Matches the first occurrence of a character that is not in the brackets.
&	Used in the *new-text* part of a substitution command, the ampersand represents the *old-text* string that matches the search pattern.
%	Used in the *new-text* part of a substitution command, the percent sign represents the *new-text* part of the last substitution command issued. This symbol is peculiar to **ed** and is not a regular expression metacharacter.

In the following example, **ed** searches for any three-character sequence ending in **at**.

```
1,$p ↵
rat
cat
turtle
cow
goat
g/.at ↵
rat
cat
goat
```

The * represents zero or more occurrences of whatever character precedes it in the search or substitution pattern. You could use the * in a substitution command to delete every unnecessary **r** in **brrroke** in the following example:

```
p ↵
brrroke
s/br*/br ↵
broke
```

Even though the purpose of the above substitution is not to change the **b**, the substitution pattern nevertheless includes the **b** before the **r**. If the pattern did not include the **b**, it would match the first character tested because that character (or any character for that matter) would constitute a zero occurrence of **r**. In the example above, **ed** would see that a pattern of **r*** matches the **b** in **brrroke**, so **ed** would perform the substitution there and quit. The result would be **brrrroke**. Remember that in a substitution, **ed** changes only the first occurrence of the pattern unless you request a global substitution with **g**.

The following example shows how the substitution would occur if the pattern were only *.

```
p ↵
brrroke
s/*/r ↵
rbrrroke
```

If you combine the period and the *, the combination will match all characters. With this combination you can replace all characters in the last part of a line:

```
p ↵
Toads are slimy, cold creatures
s/are.*/are wonderful and warm ↵
Toads are wonderful and warm
```

The **.*** can also replace all characters between two patterns.

```
p ↵
Toads are slimy, cold creatures
s/are.*cre/are wonderful and warm cre ↵
Toads are wonderful and warm creatures
```

If you want to insert a word at the beginning of a line, use the ^ (caret) in the search or substitution pattern to represent the beginning of the line. The next example places the word **all** at the beginning of each line:

```
1,$p ↵
creatures great and small
things wise and wonderful
things bright and beautiful
1,$s/^/all / ↵
1,$p ↵
all creatures great and small
all things wise and wonderful
all things bright and beautiful
```

The $ is useful for adding characters at the end of a line or a range of lines:

```
1,$p ↵
I love
I need
I use
The IRS wants my
1,$s/$/ money. ↵
1,$p ↵
I love money.
I need money.
I use money.
The IRS wants my money.
```

In the preceding example, **ed** does not treat the **.** at the end of **money.** as a special character because it does not appear as part of the substitution pattern; it appears in the replacement text. In replacement text, the only characters that have special meaning are the ampersand (&), the percent sign (%), and the backslash (\).

If you need to include a special character in a search or substitution pattern but you do not want **ed** to interpret it as one, you need to escape the special character by preceding it with \. For example, the following screen shows how to take away the special meaning of the period:

```
p ↵
This is wonderful.
s/\./! ↵
This is wonderful!
```

You can use the same method with a \ itself. If you want to treat a \ as a normal text character, precede it with a \. For example, if you want to replace the \ symbol with the word backslash, use the substitute command line shown in the following screen:

```
1,2p ↵
This chapter explains
how to use the \.
s/\\/backslash ↵
how to use the backslash.
```

If you want to add text without changing the rest of the line, the **&** provides a useful shortcut. The **&** repeats the old text in the replacement pattern, so you do not have to type the pattern twice. For example:

```
p ↵
The neanderthal skeletal remains
s/thal/& man's/ ↵
p ↵
The neanderthal man's skeletal remains
```

Ed remembers the last search or substitution pattern you used, supplying it if you omit the pattern from a search or substitute command. **Ed** also remembers the last replacement pattern (the *new-text* in a substitution command) that you used, but **ed** does not automatically supply it if you omit a replacement pattern from a substitution command. If you include the percent sign (**%**) in the replacement pattern of a substitution command, **ed** substitutes the replacement pattern you most recently used.

For example, to change the word **money** to the word **gold**, repeat the last substitution from line 1 on line 3, but not on line 4.

```
1,$n ↵
1       I love money
2       I need food
3       I use money
4       The IRS wants my money
1s/money/gold ↵
I love gold
3s//% ↵
I use gold
1,$n ↵
1       I love gold
2       I need food
3       I use gold
4       The IRS wants my money
```

Ed automatically remembers the word `money` (the old text to be replaced), so you do not have to repeat that string between the first two delimiters. The **%** sign tells **ed** to use the last replacement pattern, **gold**.

When you use brackets in a search or substitution pattern, **ed** tries to match the first occurrence of any of the characters enclosed in the brackets and substitute the specified old text with new text. The brackets can be at any position in the pattern to be replaced.

In the following example, **ed** changes the first occurrence of the numbers `6, 7, 8,` or `9` to `4` on each line in which it finds one of those numbers:

```
1,$p ↵
Monday          33,000
Tuesday         75,000
Wednesday       88,000
Thursday        62,000
```

```
1,$s/[6789]/4 ↵
Monday           33,000
Tuesday          45,000
Wednesday        48,000
Thursday         42,000
```

The next example deletes the **Mr** or **Ms** from a list of names:

```
1,$p ↵
Mr Arthur Middleton
Mr Matt Lewis
Ms Anna Kelley
Ms M. L. Hodel
1,$s/M[rs] // ↵
1,$p ↵
Arthur Middleton
Matt Lewis
Anna Kelley
M. L. Hodel
```

If a ^ is the first character in brackets, **ed** interprets it as an instruction to match characters that are not within the brackets. However, if the caret is in any other position within the brackets, **ed** interprets it literally, as a caret.

```
1,$p ↵
grade  A   Computer Science
grade  B   Robot Design
grade  A   Boolean Algebra
grade  D   Jogging
grade  C   Tennis
1,$s/grade [^AB]/grade A ↵
1,$p ↵
grade  A   Computer Science
grade  B   Robot Design
grade  A   Boolean Algebra
grade  A   Jogging
grade  A   Tennis
```

Whenever you use special characters as wildcards in the text to be changed, remember to use a unique pattern of characters. In the above example, if you had used only

```
1,$s/[^AB]/A ↵
```

you would have changed the **g** in the word **grade** to **A**.

Rearranging Text

You have now learned to address lines, create and delete text, and make substitutions. **Ed** has one more set of versatile and important commands. You can move, copy, or join lines of text in the editing buffer. Table 3-9 shows the commands that move text.

Table 3-9 Summary of Commands for Moving Text

Command	Definition
m	Moves the specified line(s) to another location in the buffer.
t	Copies (transfers) the specified line(s) to another location in the buffer.
j	Joins the current line with the line that follows it.

Moving Lines (m)

The **m** command allows you to move blocks of text to another place in the buffer. The general format is

[*address1* [*,address2*]] **m** [*address3*]

where:

address1 [*,address2*] defines the range of lines to be moved. If only one line is moved, only *address1* is given. If no address is given, the current line is moved.

m is the move command.

address3 defines the line after which you want the text to appear.

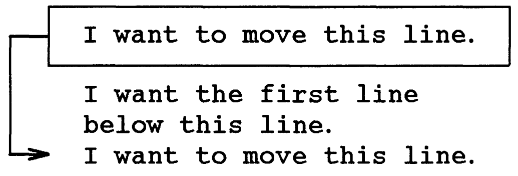
Try the following example to see how the command works. Create a file that contains these three lines of text:

```
I want to move this line. ↵
I want the first line ↵
below this line. ↵
```

Type the following command.

1m3 ↵

Ed will move line 1 below line 3.



```
I want to move this line.

I want the first line
below this line.
I want to move this line.
```

The result appears as follows.

```
1,$p ↵
I want to move this line.
I want the first line
below this line.
1m3 ↵
1,$p ↵
I want the first line
below this line.
I want to move this line.
```

If you want to move a paragraph of text, have *address1* and *address2* define the range of lines of the paragraph.

In the following example, a block of text (lines 8 through 12) is moved below line 65. Notice the **n** command that prints the line numbers of the buffer.

```
8,12n ↵
8       This is line 8.
9       It is the beginning of a
10      very short paragraph.
11      This paragraph ends
12      on this line.
64,65n ↵
64      Move the block of text
65      below this line.
8,12m65 ↵
59,65n ↵
59      Move the block of text
60      below this line.
61      This is line 8.
62      It is the beginning of a
63      very short paragraph.
64      This paragraph ends
65      on this line.
```

When *address3* is 0, as in the following example, the lines are placed at the beginning of the buffer.

```
3,4m0 ↵
```

Copying Lines (t)

The copy command **t** (transfer) acts like the **m** command except that the block of text is not deleted at the original address of the line. A copy of that block of text is placed after a specified line of text. The general format of the command line is also similar:

[*address1* [*,address2*]] **t** [*address3*]

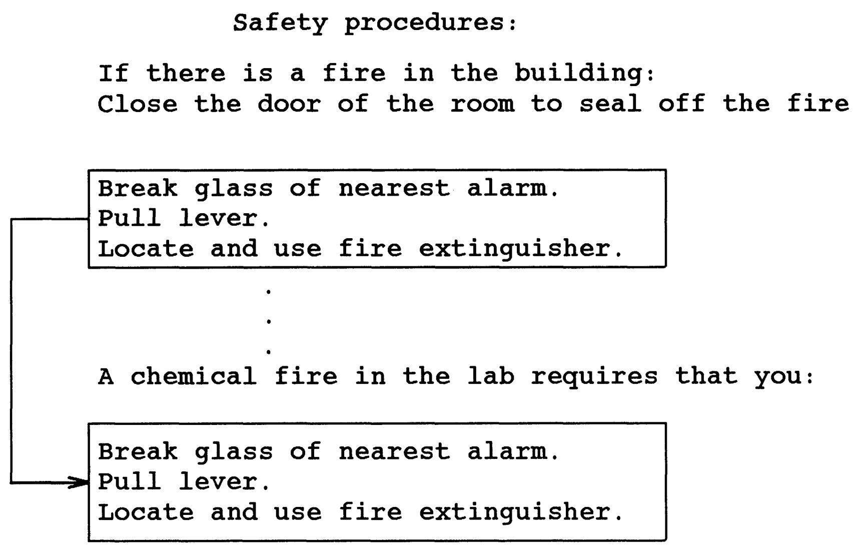
where:

address1 [*,address2*] defines the range of lines to be copied. If only one line is copied, only *address1* is given. If no address is given, the current line is copied.

t is the copy command.

address3 defines the line after which you want the text to appear.

The next example shows how to copy three lines of text below the last line.



```
        Safety procedures:

    If there is a fire in the building:
    Close the door of the room to seal off the fire

    Break glass of nearest alarm.
    Pull lever.
    Locate and use fire extinguisher.
              .
              .
              .
    A chemical fire in the lab requires that you:

    Break glass of nearest alarm.
    Pull lever.
    Locate and use fire extinguisher.
```

The commands and the responses to them are displayed in the next screen. Again, the **n** command displays the line numbers:

```
5,8n ↵
5       Close the door of the room to seal off the fire.
6       Break glass of nearest alarm.
7       Pull lever.
8       Locate and use fire extinguisher.
30n ↵
30      A chemical fire in the lab requires that you:
6,8t30 ↵
30,$n ↵
30      A chemical fire in the lab requires that you:
31      Break glass of nearest alarm
32      Pull lever
33      Locate and use fire extinguisher
6,8n ↵
6      Break glass of nearest alarm
7      Pull lever
8      Locate and use fire extinguisher
```

The text in lines 6 through 8 remains in place. A copy of those three lines is placed after line 30.

Joining Contiguous Lines (j)

The **j** command joins the current line with the following line. The general format is:

[*address1* [*,address2*]] **j**

The next example shows how to join several lines together. An easy way of doing this is to display the lines you want to join using **p** or **n**.

```
1,2p ↵
Now is the time to join
the team.
p ↵
the team.
1p ↵
Now is the time to join
j ↵
p ↵
Now is the time to jointhe team.
```

Notice that there is no space between the last word of the first line "`join`" and the first word of the second line "`the.`" You must put a space between them by using the **s** command.

Using Other Files

Once you have invoked **ed** to edit a particular file, you are not limited to using just that one file. Table 3-10 summarizes the commands that give you access to other files:

Table 3-10 Summary of Commands for Manipulating Files

Command	Definition
r	Inserts the contents of a file into the buffer.
w	Writes lines to a file, overwriting the contents of the file if it already exists.

Reading in a File (r)

You use the **r** command to insert text from a file into the buffer. The general format for the read command is:

[*address1*] **r** *filename*

where:

address1 defines the line after which you want the read text to appear. If you do not supply *address1,* **ed** appends the read text to the end of the buffer.

r is the read command.

filename is the name of the file that will be copied into the editing buffer.

Using the example from the write command, the next screen shows a file being edited and new text being read into it.

```
1,$n ↵
1               March 17, 1986
2       Dear Michael,
3       Are you free later today?
4       Hope to see you there.
3r memo ↵
86
3,$n ↵
3       Are you free later today?
4       There is a meeting in the
5       green room at 4:30 P.M. today.
6       Refreshments will be served.
7       Hope to see you there.
```

Ed responds to the read command with the number of characters in the file being added to the buffer (in the example, **memo**).

Writing Lines to a File (w)

The **w** command writes text from the buffer into a file. If the file already exists, the **w** command overwrites its contents. The general format is

[*address1* [*,address2*]] **w** [*filename*]

where:

address1 [*,address2*] defines the range of lines to be placed in another file. If you do not use *address1* or *address2,* the entire buffer is written into the file.

w is the write command.

filename is the name of the file that will contain the copied text.

In the following example the body of a letter is saved in a file called **memo**, so that it can be sent to other people.

```
1,$n ↵
1              March 17, 1986
2       Dear Kelly,
3       There will be a meeting in the
4       green room at 4:30 P.M. today.
5       Refreshments will be served.
3,5w memo ↵
91
```

The **w** command places a copy of lines 3 through 5 into a new file called **memo**. **Ed** responds with the number of characters in the new file.

The **w** command overwrites existing files; it replaces the current contents of the file with the new block of text. If, in our example, a file called **memo** had existed before we had written our new file to that name, the original file's contents would have been lost.

You cannot write additional lines to the file **memo**. If you try to add line 1, the existing lines (3 through 5) will be erased and the file will contain only line 1.

Saving the Buffer Contents in a File (w)

As we discussed earlier, during an editing session, the system holds your text in a temporary storage area called a buffer. When you have finished editing, you can save your work by using the **w** command just described to copy the buffer contents to a file.

NOTE: It is a good idea to write the buffer text into your file frequently. If an interrupt occurs (such as an accidental loss of power to the system), you may lose the material in the buffer, but you will not lose the copy written to your file.

To write your text to a file, enter the **w** command. You do not need to specify a file name; simply type **w** and press the New Line key. If you have just created new text, **ed** creates a file for it with the name you specified when you entered the editor. If you have edited an existing file, the **w** command writes the contents of the buffer to that file by default.

When **ed** writes the buffer to a file, it reports at the bottom of the screen the number of characters written. When the editor reports the number of characters in this way, the write command has succeeded.

For more information on the **w** command and writing text to files, see the section "Writing Lines to a File."

Quitting ed (q)

After you have saved the buffer contents, terminate the editing session and return to the shell by typing **q** (for quit).

```
w ↵
85
q ↵
$
```

The system responds with a shell prompt. At this point the editing buffer vanishes. If you have changed the text in the editing buffer but have not saved it by executing the **w** command, **ed** warns you by responding with an error. You then have a chance to write the buffer to a file before quitting. In the following example, Help mode is off, so the error is only ?.

```
q ↵
?
w ↵
85
q ↵
$
```

If, instead of writing the buffer to a file, you type **q** a second time, the editing buffer's contents vanish and **ed** returns you to the shell without saving your changes. Your original file, if there was one, remains unchanged.

As an alternative to **q**, you can use the **Q** command. The **Q** command is somewhat more risky than its lowercase counterpart because it tells **ed** to terminate even if you have not saved your changes. If you issue **Q** without first writing the editing buffer to a file, **ed** does not stop to display a warning — it terminates and returns you to the shell, and the editing buffer vanishes.

Other Useful Commands and Information

There are some other commands and a special file that may be useful to you during editing sessions. Table 3-11 summarizes these commands.

Table 3-11 Summary of Other Useful Commands

Command	Definition
P	Turns prompting on and off.
h	Display a short error message for the preceding diagnostic ?.
H	Turn on help mode. An error message will be given with each diagnostic ?. Typing **H** a second time turns off help mode.
l	Display nonprinting characters in the text.
f	Display the current file name.
f *newfile*	Change the current file name associated with the editing buffer to *newfile*.
!*cmd*	Temporarily escape to the shell to execute a shell command *cmd*.

In addition, you may find it useful knowing about the **ed** special work file, **ed.hup**, which you can use in recovering an interrupted editing session. A discussion of the **ed.hup** file appears at the end of this section.

The use of ( ) and { } are discussed in the **ed**(1) man page of the *User's Reference for the DG/UX™ System.*

Prompting (P)

By default, **ed** does not issue any kind of prompt to show that it has finished the last command and is waiting for you to issue the next. To enter prompt mode, issue the **P** command. The **ed** prompt is the asterisk (*). Pressing **P** a second time turns off prompt mode.

```
↵
This is the first line of text.
P ↵
*s/text/brilliance ↵
This is the first line of brilliance.
*P ↵
s/brilliance/nonsense ↵
This is the first line of nonsense.
```

Help Commands (h)

When you type a command resulting in an error, **ed** responds with a question mark (?). If you would like more detailed information when errors occur, use the help commands. There are two help commands.

The **h** command displays a short error message explaining the most recent ?. The **H** command not only displays a message explaining the last ?, but it also turns on help mode so that a short error message is displayed every time the ? appears. Turn off help mode by issuing **H** a second time.

You know that if you try to quit **ed** without writing the changes in the buffer to a file, you will get a ?. Do this now. When the ? appears, type **h**:

```
q ↵
?
h ↵
warning: expecting 'w'
```

The ? is also displayed when you specify a new filename on the **ed** command line. Give **ed** a new filename. When the ? appears, type **h** to find out what the error message means.

```
ed newfile ↵
? newfile
h ↵
cannot open input file
```

This message means one of two things: either there is no file called **newfile** or there is such a file but **ed** is not allowed to read it.

As explained earlier, the **H** command responds to the ? and then turns on the help mode of **ed**, so that **ed** gives you a diagnostic explanation every time the ? is displayed. To turn off help mode, type **H** again. The next screen shows **H** being used to turn on help mode. Sample error messages are also displayed in response to some common mistakes:

```
$ ed newfile ↵
?newfile
H ↵
cannot open input file
/hello ↵
?
illegal suffix
1,22p ↵
?
line out of range
a ↵
I am appending this line to the buffer. ↵
. ↵
s/$ tea party ↵
?
illegal or missing delimiter
,$s/$/ tea party ↵
?
unknown command
H ↵
q ↵
?
h ↵
warning: expecting 'w'
```

These are some of the most common error messages that you may encounter during editing sessions:

`illegal suffix`
Ed cannot find an occurrence of the search pattern **hello** because the buffer is empty.

`line out of range`
Ed cannot print any lines because the buffer is empty or the line specified is not in the buffer.

A substitution command could return the following errors.

`illegal or missing delimiter`
The delimiter between the old text to be replaced and the new text is missing.

`unknown command`
address1 was not typed in before the comma; **ed** does not recognize **,$**.

Displaying Nonprinting Characters (l)

If your text contains nonprinting characters such as tabs or a control character, you may want to use the **l** (list) command. The general format for the **l** command is the same as for **n** and **p**.

[*address1* [*,address2*]] **l**

where:

address1 [*,address2*] defines the range of lines to be displayed. If no address is given, the current line will be displayed. If only *address1* is given, only that line will be displayed.

l is the command that displays the nonprinting characters along with the text.

When you use the **l** command, tabs appear as a **>** (greater than) character. Control characters appear as a \nnn sequence, when *nnn* is the octal ASCII value for the control character. For example, **ed** displays Ctrl-G as \007, which is the corresponding octal ASCII code.

Type in two lines of text that contain a Ctrl-G and a Tab. Then use the **l** command to display the lines of text on your terminal.

```
a ↵
Add a Ctrl-G (control-g) to this line. ↵
Add a Tab (tab) to this line. ↵
. ↵
1,2l ↵
Add a \007 (control-g) to this line. ↵
Add a > (tab) to this line. ↵
```

Your terminal's bell may sound when it prints the Ctrl-G because Ctrl-G is the ASCII code for the bell character.

Displaying and Changing the Current Filename (f)

Ed associates a filename with the editing buffer. When you use the **w** command without a filename argument, **ed** writes the buffer to this filename. By default, this file is the one you named when you invoked **ed**. To display the filename associated with the editing buffer, use the **f** command. You can also use the **f** command to change the current default filename. You do this by issuing the **f** command with a filename argument. After changing the current filename, you can write the buffer to it by issuing the **w** command without an argument.

The format for displaying the current filename is **f** alone on a line:

f

To associate the contents of the editing buffer with a new filename, use this general format:

f *newfilename*

This command does not change the name of the original file (if it exists); it causes **ed** to create a new file or overwrite the contents of the file if it already exists.

Escaping to the Shell (!)

There may be times when you would like to issue a shell command without leaving **ed**. For example, you may want to make sure a file does not already exist before using the **w** command to write text to it. The **!** allows you to issue a shell command without terminating **ed**.

The general format for the escape sequence is

!*shell command line*

When you type the **!** as the first character on a line, the shell command must follow on that same line. The shell's response to your command will appear on the succeeding lines. When the command has finished executing, a ! will be appear alone on a line. This means that that you are back in the editor at the current line.

For example, if you want to return to the shell to find out the correct date, type **!** and the shell command **date**.

```
p ↵
This is the current line
!date ↵
Tue  Apr 1  14:24:22  EST  1986
!
p ↵
This is the current line.
```

You can even use **!** to execute **sh** or **csh**, according to your preference, so you can issue numerous commands. Another way to issue more than one command with the **!** command is with the semicolon (;). For more information on this use of the semicolon, see *Using the DG/UX™ System*.

Recovering from System Interrupts

If, while you are using **ed**, there is an interrupt to the system or your process hangs, the system tries to save the contents of the editing buffer in a special file named **ed.hup**. Later you can retrieve your text from this file in one of two ways. First, you can use a shell command to move **ed.hup** to another file name, such as the name the file had while you were editing it (before the interrupt). Second, you can enter **ed** and use the **f** command to rename the contents of the buffer. An example of the second method is shown in the following screen:

```
ed ed.hup ↵
928
f myfile ↵
myfile
```

If you use the second method to recover the contents of the buffer, be sure to remove the **ed.hup** file afterward.

End of Chapter

Chapter 4
Using the Batch Editor: sed

Sed is a noninteractive editor used for editing a copy of the text from a file or standard input (the terminal screen, input redirected from another source, or input piped through other system commands). The edited version goes to the screen (or standard output) by default or you can redirect it to a file.

How sed Processes Input

Sed follows these steps to process input.

1) **Sed** reads a line from either standard input (the terminal, redirection, or piping) or from an input file into an editing buffer called the pattern space.

2) **Sed** reads the first command (from the command line or a script file), and if the address in the command selects the pattern space, **sed** edits it.

3) **Sed** then reads the next command. If the address in the command selects the pattern space, **sed** edits it.

4) If there are more commands, **sed** repeats the previous step for each command. **Sed** then sends the results to standard output.

5) If there is more input, **sed** repeats the entire previous procedure.

The sed Command Format

Sed provides a convenient way to edit text that is located in an input file using a **sed** command that you type on the command line or multiple editing commands contained in a script file. The general form for a **sed** command follows:

sed [**−n**] [**−e** '*sed-command*' [...]] [**−f** *script-file*] [*input-file*]

where:

−n (no print); **sed** will not write the edited lines to standard output (which is the default). As an alternative, you can use the **n** argument to the comment (#) line to produce the same effect (refer to a later section on "Using Comments in a sed Script" for more information). The **p**

command (introduced in Table 4-2) can be used to print explicit lines.

−e (each command); if you use multiple **sed** commands on the command line, each one must be preceded by the **−e** option. If you supply another option to the **sed** command (such as **−n**) it must precede the **−e** option. A space separates the **−e** option from its argument. If you use only one command, the **−e** option is unnecessary.

−f (file); for **sed** scripts only, precedes the name of the file containing the **sed** script. It tells **sed** to read its program from *script-file*. A space separates the **−f** option from its argument.

input-file The *input file*(s) provides the input to the **sed** script. If you do not specify an input file, it is assumed that **sed** will receive standard input from the terminal.

A Sed Command Line Versus a Script

You may prefer to write and execute a **sed** command from the shell if the problem you're solving is simple (only one line is necessary), and if you plan to use it only for the current log-in session. If the problem you're solving is fairly complicated (it will take more than one line), and if you plan to use it repeatedly, then you'll probably want to write a script.

NOTE: The Bourne shell permits you to write multi-line **sed** commands interactively through its secondary prompt (>) facility. You will need to escape each new-line with a backslash (\) except the final new-line, which terminates the **sed** command.

Addressing Input Lines

By default, **sed** searches an input file starting at the first line. The last line is signified by a dollar sign ($). There are two ways to request a line:

- By absolute line number(s).
- By context (with regular expression pattern matching).

Information on regular-expression pattern matching is given in Appendix A.

Lines can be represented using a single address or by using two addresses to signify a range of text to be affected by the editing command(s). The first address (represented as *x*) specifies the first line in a range; the second address (represented as *y*), the final line in the range. Table 4-1 gives examples of line addressing.

Table 4-1 Addressing Methods

Format	Example	Description
x	**5**	Addresses one absolute line number
x,y	**5,9**	Addresses a range of lines signified by two absolute line numbers.
/pattern/	**/^Dispatch/**	Addresses lines containing a single pattern.
/pattern/,/pattern/	**/^Dispatch/,/Cabs/**	Addresses a range of lines identified by two patterns.
/pattern/,x	**/^Dispatch/,9**	Addresses a range of lines identified by a pattern and a single absolute line number.
x,/pattern/	**1,/^Dispatch/**	Addresses a range of lines identified by a single absolute line number and a pattern.
*any-addressing-method***!**	**1,2!**	Addresses all lines except those specified in the range. If you use the C shell and you specify an editing command on the command line, you will have to escape the exclamation point with a backslash (**\!**) to prevent the C shell from interpreting it as the history recall character.

Sed Editing Operations

Table 4-2 lists the types of editing operations **sed** performs.

Table 4-2 Sed Editing Commands

Editing Operation	Command	Definition
print addressed line	**p**	Searches for the address (which is an implicit pattern or an absolute line number) and prints to standard output.
line number	**=**	Places the current line number in standard output as a line.
append	**a**	Appends new text after the addressed line in standard output.
insert	**i**	Inserts new text before the addressed line in standard output.
change	**c**	Changes addressed text to new text in standard output.
delete	**d**	Deletes addressed line(s) from pattern space.
substitute	**s**	Substitutes pattern(s) within addressed line(s) with replacement string in pattern space.
write	**w**	Writes addressed line(s) to file.
read	**r**	Reads text from a file to an addressed line in standard output.
quit	**q**	Quits searching after the first occurrence of a pattern match is found in pattern space
show control characters as ASCII codes	**l**	Represents control character as its two-digit ASCII code octal equivalent.
convert case	**y**	Substitutes the search string with a replacement string in the alternate case. The strings must be of equal length.
groups commands	**{}**	Surrounds a set of commands to be performed on only the addressed lines.
branching with label	**b**	Tests for an address match for which control jumps to a labelled set of instructions. Control then goes to the end of the script.

(continued)

Table 4-2 Sed Editing Commands

Editing Operation	Command	Definition
testing with label	**t**	Tests whether or not a successful substitution occurs on the current line. If so, control jumps to a labelled set of instructions. Control then goes to the end of the script.
reads next line into pattern space	**N**	Reads the next line of input from an input file and appends it to the current pattern space.

(concluded)

Using Blank Lines in a sed Script

You can insert blank lines throughout a **sed** script to make it easier to read. These lines will not be interpreted.

Using Comments in a sed Script (#)

You can place a single comment line in a **sed** script as a documentation aid. All text on the line following a **#** (pound) sign is ignored by the script. The **#** must be positioned at the beginning of only the first line. An error message will be displayed, and the script will not execute if multiple comments are in the script. The general format of the comment line follows:

#[**n**] *comment*

where:

indicates a comment.

The **n** causes **sed** to suppress all lines from standard output. This argument, which must appear immediately after the **#**, has the same effect as the **sed** command's **−n** option. If you use the **n** argument after the **#** comment delimiter, you do not need to include the **−n** option on the **sed** command line. If you do not use the **n** argument or the **−n** option, **sed** sends the edited input to standard output.

comment is the text serving as documentation.

Sample Input File

An input file named **text** will be used in this chapter to show the effect of the editing commands. It follows:

```
$ cat text ↵
Dispatch to all Cab Drivers:
If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
I can hear those meters ticking now.
```

Printing Addressed Line (p)

The print command (**p**) prints the addressed line that it matches. The general format for the print command follows:

[*address* [,*address*]] **p**

An example of using the print command follows:

```
$ sed '/^I/p' text ↵
```

This simple script searches for all occurrences of a pattern beginning with "I" that are located at the beginning of the line.

The standard output produced is sent to your terminal screen:

```
Dispatch to all Cab Drivers:
If you plan to be in the vicinity
If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
I can hear those meters ticking now.
I can hear those meters ticking now.
```

Notice that the edited version of every input line is sent to standard output by default. The addressed lines (those beginning with "I") are printed twice to standard output.

To eliminate the redundancy of the selected addressed lines in the standard output, you can use the **−n** argument on the command line to "not print" all lines. Only the lines that the **p** command matches are printed.

```
$ sed −n '/^I/p' text ↵
```

The standard output produced is sent to your terminal screen:

```
If you plan to be in the vicinity
I can hear those meters ticking now.
```

Numbering Lines (=)

The line numbering command (**=**) places the current line number in standard output as a line.

The general format for the line number command follows:

[*address*]**=**

An example of printing the addressed patterns with corresponding line numbers follows:

```
$ sed -n -e '/^I/=' -e '/^I/p' text ↵
```

The standard output produced is sent to your terminal screen:

```
2
If you plan to be in the vicinity
5
I can hear those meters ticking now.
```

Appending Text (a)

The append command (**a**) appends one or more lines of text to the addressed line or to each line (if no address is given) in standard output. Each append command accepts only one address (not a range). Since appended text is sent to standard output, it cannot be edited by the **sed** commands, which operate on the pattern space only. The general format for the append command follows:

[*address*]**a\**
text \
text \
text \
•
•
•
text

where:

address specifies a single absolute line number or a pattern in the *input-file* to which the new text is appended in the output. Only a single address can be used; refer to the section "Addressing Input Lines" for examples.

The **a** command (append) adds the specified text to the addressed line in standard output.

Each line, except the final one, ends with a backslash (\\), which preserves the effect of the new-line. When the script executes, the specified text is appended exactly as specified by the script. Thus, the escaped new lines perform the appropriate line breaks. Without the backslash to preserve the effect of the new-line, the script would interpret the first new line as a signal to end the script. Thus, no explicit new lines would be performed.

The *text* comprises the new words that you append in the output at the specified address.

The final new line requires no escape because it terminates the appended text. If you press the New Line key alone on a line to terminate the append command, an extra blank line will appear in your output.

An example of a **sed** script named **appender** follows:

```
$ cat appender ↵
/business/ a\
THERE IS A RECORD SELLOUT.
```

This script will locate the line containing the pattern "`business`" and will append the specified text on the following line. To run this script, you type:

```
$ sed —f appender text ↵
```

The standard output produced is sent to your terminal screen:

```
Dispatch to all Cab Drivers:
If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
THERE.IS A RECORD SELLOUT.
I can hear those meters ticking now.
```

Inserting Text (i)

The insert command (i) inserts one or more lines of text before the addressed line in the pattern space or before each line in the pattern space if no address is given. The insert command accepts only one address. Since inserted text is sent to standard output, it cannot be edited by the **sed** commands that operate on the pattern space only.

The general format for the insert command follows:

[*address*]**i**
text ****
text ****
text ****
.
.
.
text

where:

address specifies the addressed line that is matched in the pattern space to which the new text is appended. Only a single address can be used; refer to the section

"Addressing Input Lines" for examples.

The **i** (insert) command inserts the specified text in the output at the given address.

The *text* comprises the new words that you append to the output at the specified address.

The final new-line requires no escape because it terminates the appended text. If you press the New Line key on a line alone to terminate append mode, an extra blank line will appear in your output.

An example of a **sed** script named **inserter** follows:

```
$ cat inserter ↵
3 i\
THERE IS A RECORD SELLOUT.
```

This script will insert the identified text on the line above the addressed line.

To run this script, you type:

```
$ sed -f inserter text ↵
```

The standard output produced is sent to your terminal screen:

```
Dispatch to all Cab Drivers:
If you plan to be in the vicinity
THERE IS A RECORD SELLOUT.
of the Central City Coliseum tonight,
you are in for big business.
I can hear those meters ticking now.
```

Changing Text (c)

The change command (**c**) replaces the addressed line(s) that is matched in the pattern space with new text. Any form of addressing can be used. Since changed text is sent to standard output, it cannot be edited by the **sed** commands that operate on the pattern space only.

The general format for the change command follows:

[*address*[,*address*]]**c\**
text \
text \
text \
•
•
•
text

where:

address specifies the range of lines matched in the pattern space on which the **sed** commands execute. The address can be expressed using a variety of methods given in the section "Addressing Input Lines."

The **c** (change) command changes the addressed text matched in the pattern space to new text that you specify.

Each line, except the final one, ends with a backslash (\), which preserves the effect of the new-line. When the script executes, the specified text is appended exactly as specified by the script. Thus, the new lines perform the appropriate line breaks. Without the backslash to escape the effect of the new-line, the script would interpret the first new line as a signal to end the script. Thus, no explicit new lines would be performed.

The *text* comprises the new words that you change the addressed text in the pattern space to.

The final new-line requires no escape because it terminates the appended text. If you press the New Line key on a line alone to terminate append mode, an extra blank line will appear in your output.

An example of a **sed** script named **changer** follows:

```
$ cat changer ↵
3 c\
YOU SHOULD GET READY FOR SOME ACTION. \
YOU'LL BE CRUISING TONIGHT.
```

This script will cause the addressed line in the pattern space to be changed to the new lines of text.

To run this script, you type:

```
$ sed —f changer text ↵
```

The standard output produced is sent to your terminal screen:

```
Dispatch to all Cab Drivers:
If you plan to be in the vicinity
YOU SHOULD GET READY FOR SOME ACTION.
YOU'LL BE CRUISING TONIGHT.
you are in for big business.
I can hear those meters ticking now.
```

Deleting Text (d)

The delete command (**d**) removes the pattern space, and program control will go to the top of the **sed** script

The general format for the delete command follows:

[*address* [,*address*]]**d**

where:

address specifies the range of lines in the pattern space on which the **sed** commands execute. The address can be expressed using a variety of methods given in the section "Addressing Input Lines."

The **d** (delete) command removes the addressed text from pattern space.

An example of a **sed** command follows:

```
$ sed '/Cab/,/Coliseum/d' text ↵
```

The standard output produced is sent to your terminal screen:

```
you are in for big business.
I can hear those meters ticking now.
```

Substituting Text (s)

The substitute command (**s**) substitutes the addressed pattern with a replacement string. The format for the substitute command follows:

[*address* [,*address*]] **s/***pattern***/***replacement-string***/**[**g**] [**p**] [**w**] [*n*]

where:

address specifies the range of lines in the pattern space on which the **sed** commands execute. The address can be expressed using a variety of methods given in the section "Addressing Input Lines."

The s (substitute) command replaces the addressed pattern with a replacement string.

A pattern can be a literal string or a regular expression to be substituted.

The replacement string is the new text to which you change the pattern.

The three slashes (/ / /) in the preceding format represent delimiters that separate the s command, the pattern, and the replacement string. You can select any character as a delimiter (such as ? or @) just as long as you use the same character for all three delimiters and the character isn't used in a pattern or the replacement string. Also, the character should not be a metacharacter. Refer to Appendix A for a list of metacharacters.

You can also tag specific sub-patterns for subsequent rearrangement in the replacement string. Furthermore, the replacement string can contain an ampersand (&), which is replaced with the matched pattern. Tagging and using the ampersand is covered in a later section in this chapter "Advanced Substitution."

Four options are available to add to the substitute command. Table 4-3 lists them.

Table 4-3 Substitute Options

Option	Description
g	**Sed** substitutes all non-overlapping occurrences of the addressed pattern in a line with the replacement string. By default, only the first occurrence of the addressed pattern in a line is substituted with the replacement string.
p	**Sed** writes all lines affected by a substitution to standard output. This option overrides the **−n** option set in the command line, which prints nothing to standard output.
n	**Sed** substitutes only the *n*th occurrence (represented as a number) of a pattern with a replacement string.
w *filename*	This option is identical to the **p** option except that it appends the edited line to *filename*. There should be a space between the **w** option and the filename.

Simple Substitution (g, p)

Examples of using **sed** commands with simple substitution follow:

```
$ sed −n '2 s/in/IN/gp' text ↵
```

The **−n** argument causes no lines to be printed to standard output. The **p** option causes only changed lines to be printed to standard output. If the **p** option is used without the **−n** argument, the changed lines are explicitly printed as well as all other lines to standard output. The **g** option substitutes multiple occurrences of a pattern on the same line.

The standard output produced is sent to your terminal screen:

```
If you plan to be IN the vicINity
```

On line 2 only, the pattern "`in`" is substituted with the replacement string "`IN`". The **g** option causes the replacement pattern to be repeated for multiple occurrences of the pattern in the same line. If the **g** option had not been used, only the first occurrence of the addressed pattern would be replaced.

For substitution operations, if you use a pattern as the address, which is identical to the pattern to be substituted, you can abbreviate the command format as follows:

*/pattern-address/***s//***replacement-string/*[*options*]

A shorthand way of expressing the *pattern-address* and the pattern to be replaced as the same is to specify the pattern to be substituted as null (no space between the slashes). An example follows.

```
$ sed −n '/Dispatch/s//MESSAGE ALERT/p' text ↵
```

The standard output produced is sent to your terminal screen:

```
MESSAGE ALERT to all Cab Drivers:
```

Substituting the nth Occurrence of a Pattern (n)

Instead of substituting all occurrences of a pattern with a replacement pattern, you may choose to restrict the replacement to only the *n*th occurrence on each line. An example follows:

```
$ sed −n 's/in/IN/2p' text ↵
```

The standard output produced is sent to your terminal screen:

```
Dispatch to all Cab Drivers:
If you plan to be in the vicINity
of the Central City Coliseum tonight,
you are in for big busINess.
I can hear those meters ticking now.
```

Notice that the first occurrence of "in" was not replaced in a line; only the second occurrence was.

Writing the Substitution to a File (w)

In addition to producing standard output on the terminal screen, you can also write it explicitly to a file using the w option. An example follows.

```
$ sed −n '2,3 s/the/THE/gpw subout' text ↵
```

In this command, the affected lines are sent not only to standard output but are written to a file named **subout**, whose contents follow.

```
If you plan to be in the THE vicinity
of THE Central City Coliseum tonight,
```

Advanced Substitution

There are two advanced substitution operations that let you do these manipulations:

- Tag multiple search patterns and rearrange them within a replacement string (tagged replacement patterns).
- Save a search pattern and substitute it within a replacement string (search pattern saves).

Tagging Patterns: the Escaped Parentheses (\(\))

With the search pattern tag you can locate and mark as many as nine patterns. In the replacement pattern, you can then rearrange the order in which the tagged patterns occur.

The command format used for substituting with tagged replacement patterns follows:

sed –n 's/\\(*tag-word1***\\)...\\(***tag-word9***\\)/***tag-num1***...***tag-num9***/p'** *inputfile*

where:

/ (the slash) is a delimiter.

\\(\\) (escaped parentheses) surround tagged patterns or substrings.

\*tag-num* causes the indicated tag to be substituted in the replacement string in the specified order.

The following example shows how to use tagged search patterns:

```
$ sed –n 's/\(Central\) \(City\)/\2 \1/p' text ↵
```

The substring "`Central`" is tagged as 1; "`City`" is tagged as 2. The replacement tags cause the contents of locations 1 and 2 to be swapped in the standard output. Spacing in this command is important.

- Do not space within delimiters.
- Do not use a space to separate the final delimiter and the first replacement string.
- Do space once between replacement tags.

The example produces this result:

```
of the City Central Coliseum tonight,
```

You can see that locations 1 and 2 were swapped in your standard output.

In addition to tagging patterns for swapping, you can also simply rearrange them with other replacement text. For example, using the same sample line of text, the following command:

```
$ sed -n 's/\(Central\) \(City\) \(Coliseum\)/Metro \2 \1 Events \3/p' text ↵
```

would produce this output:

```
   of the Metro City Central Events Coliseum tonight,
```

The replacement text includes the word "Metro", the second tag "City", the first tag "Central", the word "Events", and finally the third tag "Coliseum".

Modifying the Matching String: the Ampersand (&)

This replacement construct is useful when you want to modify (rather than substitute entirely) the search pattern with a replacement string. The ampersand (&) stores the pattern found so that you can recall and place it in the replacement string.

An example of the text to be operated on follows:

```
   you are in for big business.
```

The following command modifies this text:

```
   $ sed -n 's/big/INCREDIBLY &/p' text ↵
```

where:

s stands for substitute, **big** is the search pattern, and **INCREDIBLY** is the replacement string. The ampersand (&) saves the search pattern which is used in the replacement string. Spacing is important here; space within the replacement string exactly as you want it to appear in the output.

The following result is produced:

```
   you are in for INCREDIBLY big business.
```

Writing Standard Output to a File (w)

The write command (**w**) writes the edited pattern space to a file. You can specify up to ten filenames for output in a **sed** script. It is similar to the use of the write option with the substitute command. The format for the write command follows:

[*address*[,*address*]] **w** *filename* where:

address specifies the range of lines in the pattern space on which the **sed** commands execute. The address can be expressed using a variety of methods given in the section "Addressing Input Lines."

The **w** (write) command redirects standard output to *filename*.

Only the addressed lines are sent to the file with the write command. An example follows:

```
$ sed '3,4 w writer-out' text ⤾
```

The entire text file is sent as standard output to your terminal screen.

The output sent to the file named **writer-out** follows:

```
$ cat writer-out ⤾
of the Central City Coliseum tonight,
you are in for big business.
```

Reading Text from a File (r)

The read command (**r**) reads the contents of a file and writes it to standard output. The format for the read command follows:

address **r** *filename*

where:

The *address* specifies the single position in *filename* to which the new text from *filename* is read in standard output. Only a single address can be used; refer to the section "Addressing Input Lines" for examples.

The **r** command reads the contents of the specified *filename*.

The *filename* is the name of the file from which the new text is read.

An example follows.

```
$ sed '/business/r extra' text ⤾
```

When **sed** reads the input line containing the address "`business`", the entire file named **extra** is read onto the line following the address. The contents of the file named **extra** follows.

```
$ cat extra ⤾
WE'LL NEED CABS AT THESE POSTS:
```

```
ELM & STEWARD BLVDS
ELK CREEK SHOPPING CENTER
ED'S BODY SHOP
GRAMBLING'S GARDENS
```

The standard output produced is sent to your terminal screen:

```
Dispatch to all Cab Drivers:
If you plan to be in the vicinity
of the Central City Coliseum tonight,
you are in for big business.
WE'LL NEED CABS AT THESE POSTS:
ELM & STEWARD BLVDS
ELK CREEK SHOPPING CENTER
ED'S BODY SHOP
GRAMBLING'S GARDENS
I can hear those meters ticking now.
```

Quitting after the First Pattern Match (q)

When searching for a pattern, you may want to match only the first occurrence of a pattern and then quit the process. The general format for the quit command (**q**) follows:

*address***q**

where:

address specifies the range of lines in the pattern space on which the **sed** commands execute. The address can be expressed using a variety of methods given in the section "Addressing Input Lines."

The **q** command means to quit.

An example of a **sed** command using quit follows:

```
$ sed '/C.*/q' text ↵
```

The pattern sought is a regular expression that refers to any string that begins with an uppercase "C" and is followed by any number of characters. After the first occurrence of the pattern is matched and printed, the process quits.

The standard output produced is sent to your terminal screen:

```
Dispatch to all Cab Drivers:
```

Displaying Nonprintable Control Characters in Standard Output (l)

The appearance of nonprintable control characters may not be apparent from just looking at an input file. If you use **sed** to manipulate input that contains nonprintable control characters, you may wish to display (or list) control characters in their two-digit ASCII (octal) representation. Refer to the first column of the ASCII characters table in the ASCII chart in *Using the DG/UX™ System* for the octal representation of control characters.

The general format for representing control characters as ASCII codes in standard output using the **l** command follows:

[*address*,[*address*]]**l**

where:

address specifies the range of lines in the pattern space on which the **sed** commands execute. The address can be expressed using a variety of methods given in the section "Addressing Input Lines."

The **l** command means to list.

An example of text containing control characters follows:

```
$ cat .exrc ↵
map #1 i\italic^[Ea\roman^[
```

This command is a function key map from an **.exrc** file (refer to the section "Using Key Maps" in Chapter 2 in this manual for more information). This key map allows you to use the F1 key to turn on the italic font for the selected word and then turn on the roman font at the end of the word.

The appearance of control characters can be confusing. To find out exactly the control characters that appear in your text, you can use this short **sed** command:

```
$ sed –n '1,$l' .exrc ↵
```

Standard output follows:

```
map #1 i\italic33Ea\roman33
```

The ^[notation in the pattern space represents an ASCII octal code of 33, which is an escape character that is generated with two keystrokes: Ctrl-V and Esc.

Converting Strings of Equal Length (y)

With the translate command (**y**), you can substitute each character of a string with the corresponding character in the replacement string. The search string and the replacement string must have equal lengths.

The format used for translating strings follows:

[*address* [,*address*]]**y/***string-1***/***string-2***/**

where:

The *address* specifies the range of lines in the pattern space on which the **sed** command executes. The address can be expressed using a variety of methods given in the section "Addressing Input Lines."

The **y** command means to translate.

The *string-1* refers to the string to be translated.

The *string-2* refers to what *string-2* is translated to.

An example of converting strings of equal length follows:

```
$ sed –n '1,$y/abc/ABC/' text ↵
```

Standard output follows:

```
DispAtCh to All CAB Drivers:
If you plAn to Be in the viCinity
of the CentrAl City Coliseum tonight,
you Are in for Big Business.
I CAn heAr those meters ticking now.
```

Using Control Structures: Advanced

There are three control structures you can use in **sed** scripts for grouping specific commands to process only the addressed lines in the pattern space. The control structures are:

- Grouping with braces.
- Simple branching.
- Conditional branching.

Grouping with Braces ({ })

Only the addressed lines will be affected by the commands enclosed in braces. The general format for the grouping with braces command follows:

> [*address*[,*address*]] **{**
> *editing-command*
> *editing-command*
> .
> .
> .
> **}**
> *more editing commands*

Commands that occur outside the braces will affect either all lines (including those lines affected by the commands set off by braces) or another set of addressed lines. Multiple sets of branched instructions can be used in a **sed** script.

An example of a **sed** script named **bracer** follows:

```
$ cat bracer ↵
# Script bracer performs restrictive substitutions
1,3 {
1d
s/vicinity/AREA/
3 a\
THERE IS A RECORD SELLOUT!
}
s/business/MONEY/
1,4s/in/IN/
s/RECORD/BIG/
```

The commands surrounded by braces affect only lines 1–3. The first line will be deleted and the pattern "`vicinity`" will be replaced by "`AREA`". A string of text is appended to line 3. The commands following the closed brace apply to the entire file, including lines 1–3. The pattern "`business`" will be replaced with "`MONEY`", and "`in`" will be replaced with "`IN`" only in lines 1–4. The final command is to substitute "`RECORD`" with "`BIG`".

To run this script, you type:

```
$ sed -f bracer text ↵
```

The standard output produced is sent to your terminal screen:

```
If you plan to be IN the AREA
of the Central City Coliseum tonight,
THERE IS A RECORD SELLOUT!
you are IN for big MONEY.
I can hear those meters ticking now.
```

All instructions except the last one performed as would be expected. The pattern "RECORD" was not replaced with "BIG" because **sed** editing commands operate on the pattern space only. The append command operates on standard output only; therefore, those lines cannot be affected by any subsequent editing commands. Also notice that the replacement of "in" with "IN" did not affect the "in" in "ticking" because that line was excluded from the address.

Simple Branching to Label (b)

Branching is used to test for an address match for which control jumps to a label that designates an optional set of commands. A label can contain up to eight characters. If no commands are associated with the label, control branches to the end of the script. Otherwise, normal program flow occurs sequentially. The format for the branching command follows:

```
[ address[,address ] ]bbranch-label
editing-command
editing-command
.
.
.
:branch-label
optional editing command for addressed lines
optional editing command for addressed lines
.
.
.
```

NOTE: The character "**b**" is considered part of the eight-character label.

When the address is encountered, control branches to the label.

An example of the **sed** script **brancher** follows:

```
$ cat brancher ↲
1,3b jump
s/business/MONEY/
s/big/INCREDIBLY &/
s/I can hear/LISTEN TO/p
:jump
/Dispatch/s//MESSAGE ALERT/
s/\(Central) \(City)/\2 \1/
```

The address is matched as soon as the program begins processing. So control is branched to the instructions labelled `jump` in which two substitutions occur on lines 1–3. After the commands in the label have executed, control returns to the script instruction that follows the jump. Lines 1–3 are not processed by the remaining commands in the script.

To run this script, you type:

```
$ sed –f brancher text ↲
```

The standard output produced is sent to your terminal screen:

```
MESSAGE ALERT to all Cab Drivers:
If you plan to be in the vicinity
of the City Central Coliseum tonight,
you are in for INCREDIBLY big MONEY
LISTEN TO those meters ticking now.
```

Conditional Branching to Label (t)

The **t** (test) command tests whether or not a successful substitution has occurred on the current line. If there has been a substitution on the current line, control branches to a label (signifying a group of commands). A label can contain up to eight characters. If no commands are associated with the label, control goes to the end of the script. Otherwise, normal program flow occurs sequentially. The format for the conditional branching command follows:

[*address* [*,address*]]s*/pattern/replacement-string/*
t*branch-label*
editing command
editing command
.
.
.
:*branch-label*
optional branch editing command
optional branch editing command
.
.
.

NOTE: The character "**t**" is considered part of the eight-character label.

An example of a **sed** script named **conditioner** follows:

```
$ cat conditioner ⤸
s/Cab Drivers/CABBIES/
t fixer
/business/ c\
EXPECT A LATE NIGHT ON THE STREETS.
/meters/ a\
BE SURE YOU TURN IN YOUR TIME \
BEFORE YOU PUNCH OUT FOR THE EVENING. \
YOUR FRIENDLY DISPATCHER, \
ROSE
:fixer
2,3d
```

Upon a successful substitution of "`CABBIES`" for "`Cab Drivers`," control branches to the label identified as `fixer`, where lines 2–3 are deleted and control goes to the bottom of the script. Otherwise, the line containing the pattern "`business`" is changed to another string. Then new text is appended to the line containing the pattern "`meters`".

To run this script, you type:

```
$ sed —f conditioner text ⤸
```

The standard output produced is sent to your terminal screen:

```
Dispatch to all CABBIES:
EXPECT A LATE NIGHT ON THE STREETS.
I can hear those meters ticking now.
BE SURE TO TURN IN YOUR TIME
BEFORE YOU PUNCH OUT FOR THE EVENING.
YOUR FRIENDLY DISPATCHER,
ROSE
```

Pattern Matching Across Lines (N)

The **N** command is useful for pattern matching across two lines. It reads the next line of input from an input file and appends it to the pattern space.

A common editing task is to search for the occurrence of a pattern followed by a new-line and to strip out the new line, thus joining lines.

A sample input file named **memo** follows:

```
$ cat memo ↵
MEMO↵
↵
FROM:↵
Della Street↵
↵
TO:↵
Professor Ruhlen↵
↵
RE:↵
Proteins↵
↵
```

NOTE: You will not see an explicit new-line (↵) at the end of each line in either your input or output. Throughout this section, new-lines are explicitly displayed so that you can see the effect of editing them.

This is a general memorandum format. Your goal is to join all lines containing a colon (:) with the next line. For example, you want to produce this result:

```
FROM:  Della Street
```

To accomplish this task, you need to perform these steps in a **sed** script:

- Search for a pattern ending with a colon (:) and keep it in the pattern space.
- Append the next line to the current line in the pattern space.
- Substitute the new-line with two spaces, which implicitly joins the two lines.

An example of a **sed** script named **patterner** follows:

```
$ cat patterner ↵
/.*:$/{
N
s/\\n/  /
}
```

A successful address match, any line ending in a colon (:), causes the execution of the instructions surrounded by braces. The matched pattern will remain in the pattern space and the **N** (next) instruction appends the next line of input to the pattern space. The third instruction substitutes the new-line (represented as **\n**) with two spaces, thus joining the two lines.

Figure 4-1 illustrates the steps in the process.

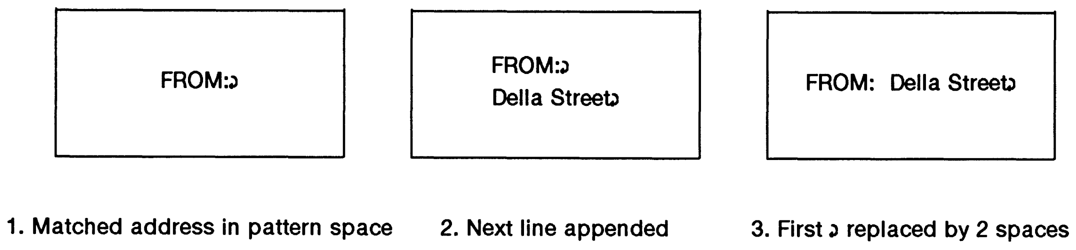

FROM:↲
FROM:↲
Della Street↲
FROM: Della Street↲
1. Matched address in pattern space
2. Next line appended
3. First ↲ replaced by 2 spaces

Figure 4-1 Steps for Pattern Matching Across Lines

To run this script, you type:

```
$ sed −f patterner memo ↲
```

The standard output produced is sent to your terminal screen:

```
MEMO

FROM:  Della Street

TO:  Professor Ruhlen

RE:  Proteins
```

There are several more **sed** commands used for manipulating the pattern space and another buffer area called the hold space. Refer to the **sed**(1) man page in the *User's Reference for the DG/UX™ System* for more information.

End of Chapter

Appendix A
Regular Expressions

Several DG/UX text-oriented commands (such as **ed**, **editread**, **ex**, **expr**, **grep**, **pg**, **sed**, and **vi**) use regular expressions, which are character patterns used to search and sometimes replace matching patterns in a file. The **egrep** and **fgrep** commands also use an additional set of metacharacters not covered here. Refer to the **grep**(1) man page for this information.

A regular expression can be expressed as either a simple string or a complex pattern containing special characters called metacharacters. For example, the regular expression "star" can match each occurrence of **star** in "**star**", "**star**t", "re**star**", or "re**star**ting". It will not match the pattern "**first arrival**" because there is a space between "t" and "a". Each of the letters in the regular expression must appear in the same sequence in the matching pattern.

Regular expression metacharacters can contain "wildcards" that locate a range of matching patterns. For example, you can use the period (.) metacharacter to match any single character. Thus, the regular expression **s.ng** will match "**sang**", "**sing**", "**song**", and "**sung**". The regular expression would also match the occurrence of **sang** in "**sang**uine" or **seng** in "**seng**i" and so on. A space could also be a matching pattern such as **s ng** in "country'**s ng**wee".

The commands that use regular expressions don't all recognize the entire set of metacharacters. All of them do follow a basic set, while many commands recognize an extended set.

Basic Set of Regular Expressions

Table A-1 summarizes the basic set of metacharacters used for regular-expression pattern matching.

Table A-1 Regular-Expression Metacharacters

Metacharacter	Definition
^	Matches a string at the beginning of a line. For example, **^dog** matches lines beginning with **dog**.
$	Matches a string at the end of a line. For example, **dog$** matches lines ending with **dog**.
.	Matches any single character except a new-line. For example, **.ilk** matches **silk** and **bilk**.
*	A single-character regular expression followed by an asterisk (*) matches zero or more occurrences of the single-character expression. For example, **ap*eal** matches **apeal**, **appeal**, and **aeal**.
\	Escapes the meaning of a metacharacter. For example, \* matches *.
[ ]	Defines a character class that matches any single character enclosed in brackets. For example, **j[oa]y** matches **joy** and **jay** but not **joay**.

Examples of each metacharacter are shown in the following sections. Throughout the examples, the matched characters will be shown in boldface type.

The Caret (^) and Dollar Sign ($)

A regular expression beginning with a caret (^) can match a string located only at the beginning of a line. Similarly, a regular expression ending with a dollar sign ($) will match a string located only at the end of a line. Table A-2 gives examples using the caret and dollar sign.

Table A-2 The Caret (^) and Dollar Sign ($) in Regular Expressions

Regular Expression	Example
^A	**A** dog fetches sticks. **A**spirin relieves headaches. **A**nd fevers too.
st$	She's the fir**st** She's the la**st** It's the steepe**st**
^$	Matches an empty line.
^who$	Matches a line containing only the string **who**.

For the **expr** command, all patterns are assumed to start at the beginning of the line.

The Period (.)

A regular expression consisting of a period matches any single character except a new-line. Table A-3 gives examples of the period (.).

Table A-3 The Period (.) in Regular Expressions

Regular Expression	Example	Explanation
.ing	**sing** **sl**ing**ing** **sting**	Matches "s" followed by "ing". Matches "l" followed by "ing". Matches "t" followed by "ing".
^..e	**The** end **12e** 14f **I e**nvy	Matches "e" preceded by "Th" at the beginning of a line. Matches "e" preceded by "12" at the beginning of a line. Matches "I" followed by a space and "e" at the beginning of a line.
.$	We like peache**s.** we have **a** Hah **!**	Matches "s" followed by "." at the end of a line. Matches a space followed by "a" at the end of a line. Matches a space followed by an exclamation mark (!) at the end of a line.

The Asterisk (*)

A regular expression containing an asterisk represents zero or more occurrences of a match of the preceding character. An asterisk preceded with a period (.*) matches any character. (A period matches any single character, and an asterisk matches zero or more occurrences of the preceding regular expression.)
Table A-4 lists examples of using the asterisk.

Table A-4 The Asterisk (*) in Regular Expressions

Regular Expression	Example	Explanation
ab*e	**ae**	Matches zero occurrences of preceding character "b".
	abe	Matches one occurrence of preceding character "b".
	abbe	Matches two occurrences of preceding character "b".
cat*	**ca**	Matches zero occurrences of preceding character "t".
	cat	Matches one occurrence of preceding character "t".
	catt	Matches two occurrences of preceding character "t".
	catman	Matches one occurrence of preceding character "t".
cat.*	**cat**	Matches zero occurrences of preceding character ".", which can be any character.
	cats	Matches one occurrence of preceding character ".", which can be any character, and some other character "s".
	catastrophe	Matches one occurrence of preceding character ".", which can be any character, and any number of other characters "astrophe".
	scatters	Matches two occurrences of preceding character ".", which can be any character, and any number of other characters "ers".

(continued)

Table A-4 The Asterisk (*) in Regular Expressions

Regular Expression	Example	Explanation
.*cat	**cat**	Matches zero occurrence of preceding character "c".
	ducat	Matches two occurrences of preceding character "c"; in this case "du".
	scatters	Matches one occurrence of preceding character "c"; in this case "s".
.cat.*s	**scats**	Matches a single character before "cat" and no characters between "cat" and "s".
	scatters	Matches a single character before "cat" and three characters between "cat" and "s".
	scatter ashes	Matches the "s" before "cat", and eight characters between "cat" and "s".
	cat is	Will not match; "cat" must be preceded by a single character.

(concluded)

The Backslash (\)

A backslash escapes a metacharacter, which retains the meaning of the literal character. Table A-5 lists examples of uses of the backslash.

Table A-5 The Backslash (\) in Regular Expressions

Regular Expression	Example	Explanation
\[weasel\]	**[weasel]**	Matches literal brackets and "weasel".
\\\\	\	Matches a backslash (\\).
*\\.c	***.c**	Matches a literal asterisk (*), a period (.), and a "c".
f.*\\.me	**file1.me**	Matches "f" followed by any number of characters "ile1" followed by ".me".
	f.me	Matches "f" followed by ".me".

The Brackets ([ ])

The brackets define a character class that matches any single character within the brackets. To specify a character class containing a range of characters or numbers, you can use a hyphen (–) between the first and last characters of the range. Metacharacters inside brackets are escaped automatically.

A caret (^) following the left bracket ([) can be used to negate what is enclosed by brackets; any pattern not specified by the regular expression inside the brackets is sought. Table A-6 lists examples of pattern matches using character classes.

Table A-6 The Square Brackets ([ ]) in Regular Expressions

Regular Expression	Example	Explanation
[Dd]og	**dog**	Matches "d" and "og".
	Dog	Matches "D" and "og".
test[1-4]	**test1**	Matches "test" and "1".
	test4	Matches "test" and "4".
	start**test2**	Matches "test" and "2".
	test4000	Matches "test" and "4".
w[ea].*	s**weater**	Matches "w", "e", and any number of following characters, which are "eater".
	waste	Matches "w", "a", and any number of following characters, which are "ste".
	we	Matches "w", "e", and any number of following characters, which are zero.
[^a-zA-Z]	**1***a!	Matches the first nonalphabetic character at the beginning of the line, which is "1".

NOTE: How you use the right bracket (]) as a member of a character class will depend on the command used. For **vi** and **ex**, the right bracket must be escaped with a backslash (\) and may be anywhere in the character class; for example, **[abc\]d]**. For **ed**, **sed**, **grep**, **expr**, **pg**, the right bracket must be the first character in the character class and does not need to be escaped; for example, **[]abcd]**.

Extended Set of Regular Expressions

Table A-7 summarizes the extended set of regular expressions for pattern matching.

Table A-7 Extended Regular-Expression Syntax

Regular Expression	Definition
\(\)	Tags a pattern for use in search and replacement operations.
&	Stores a search pattern for later use in a replacement pattern.
\<	Matches a pattern only at the beginning of a word.
\>	Matches a pattern only at the end of a word.
\{ \}	Matches a specific number of occurrences of a pattern.

The Escaped Parentheses (\(\))

Only **ed**, **ex**, **grep**, and **vi** use the escaped parentheses.

The escaped parentheses are used for tagging patterns that you can refer to by number. You tag a pattern by preceding it with **\(** and following it with **\)**. Thus, the first tagged pattern is referred to as **\1**; the second, **\2**, and so on. As many as nine patterns can be tagged for one operation. An example follows:

```
\(Central\) City \1
```

The pattern `Central` is tagged so `\1` represents `Central`. Thus, the regular expression `Central City Central` is matched. As another example, you can repeat the tag as often as needed:

```
\(Central\) \(City\) \1 \2 \1
```

This time `City` is also tagged (as `2`) and this regular expression is matched: `Central City Central City Central`.

An example of using tags in replacement patterns in **vi** follows:

```
:%s/\(Central\) \(City\)/\2 \1/
```

The percent sign (`%`) means that the entire buffer is searched for the pattern `Central City`. `Central` is tagged as 1, and `City` is tagged as 2. The replacement tags cause the contents of tags `1` and `2` to be swapped, thus producing `City Central` throughout the file.

In addition to tagging patterns for swapping, you can also rearrange them with other replacement text. A **vi** example follows:

```
:%s/\(Central\) \(City\) \(Coliseum\)/Metro \2 \1 Events \3/
```

The output produced by this example might be `Metro City Central Events Coliseum`.

The Ampersand (&)

Only **ed**, **ex**, **sed**, and **vi** use the ampersand.

This replacement construct is useful when you want to modify (rather than substitute entirely) the search pattern with a replacement pattern. The ampersand stores the pattern found so that you can recall and place it in the replacement pattern. A **vi** example follows:

```
:%s/big/INCREDIBLY &/
```

Each match of the pattern **big** is stored in **&** in the replacement pattern. An example of the output would be: `INCREDIBLY big`.

The Escaped Angle Brackets (\< \>)

Only **ex** and **vi** use escaped angle brackets.

The angle brackets are used for matching the beginning of a word (**\<**) and the end of a word (**\>**). Table A-8 lists examples of escaped angle brackets:

Table A-8 The Escaped Angle Brackets (\< \>) in Regular Expressions

Command	Locates	Produces
:s/\<driv/fall/g	**driven**	**fallen**
:s/ing\>/ers/g	singing	**singers**

The Escaped Braces (\{ \})

Only **ed**, **expr**, **grep**, **pg**, and **sed** use the escaped braces.

The braces signify a match of a specific number of occurrences of a pattern. Table A-9 lists three variations of the escaped braces:

Table A-9 Variations of Escaped Braces (\{ \})

Variation	Explanation
pattern\{*n*\}	Matches *n* occurrences of the preceding pattern.
pattern\{*n*,\}	Matches at least *n* occurrences of the preceding pattern.
pattern\{*n*,*m*\}	Matches any number of occurrences between *n* and *m* where *n* and *m* are numbers between 0 and 255.

Table A-10 shows examples of escaped braces:

Table A-10 The Escaped Braces (\{ \}) in Regular Expressions

Regular Expression	Example	Explanation
t\{2\}y	**tty**	Matches only two occurrences of **t**.
t\{4,\}y	**ttttty** **tttttty** **ttttttty**	Matches at least 4 occurrences of **t**.
t\{4,5\}y	**ttttty** **tttttty**	Matches between 4 and 5 occurrences of **t**.

An example of a **grep** command follows:

```
$ grep "t\{4,\}" parts
```

The output might include: `ttttty`, `tttttty`, `ttttttty`, and so on.

End of Appendix

Index

Note: Boldfaced page numbers (e.g., **1-5**) indicate definitions of terms or other key information.

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

      Send your order form with payment to: Data General Corporation
      ATTN: Educational Services/TIPS G155
      4400 Computer Drive
      Westboro, MA 01581–9973

   b) TELEPHONE – Call TIPS at (508) 870–1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1–4 Units	$5.00
5–10 Units	$8.00
11–40 Units	$10.00
41–200 Units	$30.00
Over 200 Units	$100.00

   If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870–1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME________	COMPANY NAME________
ATTN:________	ATTN:________
ADDRESS________	ADDRESS (NO PO BOXES)________
CITY________	CITY________
STATE________ ZIP____	STATE________ ZIP____

Priority Code __________ (See label on back of catalog)

Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) | Title | Date | Phone (Area Code) | Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

ORDER TOTAL	
Less Discount See B	–
SUB TOTAL	
Your local* sales tax	+
Shipping and handling – See A	+
TOTAL – See C	

A SHIPPING & HANDLING

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)

☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 – $149.99	0%
$150 – $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) ________

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is________. (Include hardcopy P.O.)

☐ Check or Money Order Enclosed

☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiration Date ☐☐☐☐

Authorized Signature ________
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508–870–1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION TECHNICAL INFORMATION AND PUBLICATIONS SERVICE TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Using the
DG/UX™
Editors

069-701036-01

Cut here and insert in binder spine pocket

DataGeneral

Data General Corporation, Westboro, Massachusetts 01580

069-701036-01