

**Text Editors and Processors
HP-UX Concepts and Tutorials**

**HP-UX
HP-UX
HP-UX
HP-UX
HP-UX
HP-UX**

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Text Editors and Processors HP-UX Concepts and Tutorials

HP Part Number 97089-90022



Copyright 1986 Hewlett-Packard Company

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Software clause in DAR 7-104.9(a).

Copyright 1980, 1984, AT&T, Inc.

Copyright 1979, 1980, 1983, The Regents of the University of California.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California.

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

August 1986...Edition 1

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Table of Contents

Edit: An Interactive Editor

Introduction	1
Session 1: Creating a Text File	2
Asking for <i>edit</i>	3
Creating Text	4
Messages from <i>edit</i>	4
Text Input Mode	4
Writing Text to Disc	5
Logging Off	6
Session 2	6
Adding More Text to the File	6
Interrupt	7
Making Corrections	7
Listing Buffer Contents	7
Finding Things in the Buffer	8
The Current Line	9
Numbering Lines (<i>nu</i>)	9
Substitute Command (<i>s</i>)	9
Another Way to List What's in the Buffer (<i>z</i>)	11
Saving the Modified Text	11
Session 3	12
Bringing Text Into the Buffer (<i>e</i>)	12
Moving Text in the Buffer (<i>m</i>)	12
Copying Lines (<i>copy</i>)	13
Deleting Lines (<i>d</i>)	13
Be Careful	14
Oops! I goofed. Now What? (<i>undo</i>)	15
More About Dot (<i>.</i>) and Buffer End (<i>\$</i>)	15
Moving Around in the Buffer (<i>+</i> and <i>-</i>)	16
Changing Lines (<i>c</i>)	16
Session 4	17
Making Commands Global (<i>g</i>)	17
More About Searching and Substituting	18
Special Characters	19
Issuing HP-UX Commands from the Editor	20
Filenames and File Manipulation	20
The File (<i>f</i>) Command	20
Reading Additional Files (<i>r</i>)	21
Writing Parts of the Buffer	21
Recovering Files	21
Other Recovery Techniques	21
Further Reading and Information	22
Using <i>ex</i>	22





Edit

An Interactive Editor

Introduction

Text editors are special computer programs that enable you to easily create, preserve, modify, and print text by use of a computer and terminal. Creating text is very much like typing on an electric typewriter. Modifying text involves telling the text editor what to add, change, insert, or delete. Text is printed by giving a command to print all or part of the text file contents. You can also provide special instructions to control the output format if you desire.

This tutorial is divided into four lessons, and assumes no prior familiarity with computers or with text editing. A series of text editing sessions lead you through the basic steps of creating and revising a text file. After scanning each lesson but before beginning the next, try the examples at a terminal to get a feeling for the actual process of text editing. Allow time for experimentation, and you can quickly learn to use a computer for writing and modifying text.

Other HP-UX features are useful besides the text editor. These features are discussed in the book *Introducing the UNIX System* as well as in other tutorials that provide a general introduction to the system. As soon as you are familiar with your terminal keyboard, its special keys, the system login procedure, and know how to correct typing errors, you are ready to start. Let's first define some terms:

- Program** A group of computer instructions that defines the sequence of steps to be performed by the computer in order to accomplish a specific task. For example, a series of steps to balance your checkbook is a program.
- HP-UX** A special type of program called an operating system that supervises the computer, peripheral devices, and all programs that use the HP-UX operating system.
- Edit** The name of the HP-UX text editor that you will be learning to use; a program that aids you in writing or revising text. *Edit* was designed for beginning users, and is a simplified version of a more extensive editor named *ex*.
- File** Each HP-UX account is allotted disc storage space for permanent storage of information such as programs, data, or text. A file is a logical collection of data (such as an essay, a program, or a chapter from a book) that is stored and maintained by a computer system. Once you create a file it is kept until you tell the system to remove it. You can create a file during one HP-UX session, log out, then return to use it at a later time. Files contain anything you choose to write and store in them. File sizes vary, depending on individual needs. One file might contain a single number, while another could contain a very long document or program. The only way to save information from one session to the next is to store it in a file where it is kept for later use.

- Filename** Filenames are used to distinguish one file from another, serving the same purpose as labels on manila folders in a file cabinet. To write or access information in a file, use the name of that file in an HP-UX command. The system automatically determines where the file is located.
- Disc** Files are stored on a thin circular disc that is coated with magnetic particles similar to magnetic recording tape. The disc may be permanently installed in a disc drive, or it may be a removable flexible disc that resembles a small phonograph record in a thin, square protective container or envelope. Information from the computer (such as your text) is recorded on the disc surface by the disc drive.
- Buffer** A temporary work space that is available to the user during a text editing session. The buffer is used to build and modify text files. Buffers are analagous to a piece of scratch paper that is discarded at the end of a session after the information it contained has been copied (written) to a permanent disc file.

Session 1: Creating a Text File

Before you can use the editor, you must first log onto the computer so HP-UX can set up communication between your terminal and the editor program. Here is a review of the standard HP-UX login procedure:

If the terminal you are using is directly linked to the computer, turn it on and press **(Return)**. If your terminal uses an acoustic coupler (or modem) and telephone line instead, turn on the terminal, dial the system-access telephone number, then, when you hear a high-pitched tone, place the telephone receiver in the acoustic coupler. If you are using a modem, consult the modem manual for procedures. Press carriage return (or the **(Return)** key) once, and await the login message:

```
:login:
```

Type your login name (which identifies you to HP-UX) on the same line as the login message, then press **(Return)**. If the keyboard on your terminal supports both uppercase and lowercase, be sure you enter your login name in lowercase. Otherwise, HP-UX assumes your terminal has only uppercase and will not recognize any lowercase letters you may type. When HP-UX types `:login:`, reply with your login name, for example `susan`:

```
:login: susan (Return)
```

(In this example, input typed by the user appears in bold face to distinguish it from information displayed by HP-UX.)

HP-UX responds with a request for a password as an additional precaution to prevent unauthorized people from using your account. The password will not appear when you type it (to prevent others from seeing it). The message is:

```
Password: _ (type your password and press (Return))
```


If any of the information you gave during the login sequence was mistyped or incorrect, HP-UX responds with:

```
Login incorrect.
:login:
```

If this happens, start over and repeat the process. When you successfully log in, HP-UX prints the message of the day and eventually presents you with a % at the beginning of a fresh line. The % is the HP-UX prompt symbol that tells you HP-UX is ready to accept a command.

Asking for *edit*

You are ready to tell HP-UX that you want to use *edit*, the text editor program. Now is a convenient time to choose a name for the text file you are about to create. To begin your editing session type `edit` followed by a space, then the filename you have selected, such as *text*. When you have completed the command, press **Return** and wait for *edit*'s response:

```
% edit text
"text" no such file or directory
:
```

If you typed the command correctly, you will now be in communication with *edit*. *Edit* has set aside a buffer for use as a temporary working space during your current editing session. It also checked to see if the file you named, *text*, already exists. As we expected, it was unable to find such a file since *text* is the name of the new file to be created. *Edit* confirms this with the line:

```
"text" No such file or directory
```

The colon on the next line is *edit*'s prompt, announcing that *edit* expects a command from you. You are now ready to create the new file.

The “Command not found” Message

Suppose you misspelled *edit* by typing *editor*. Your request would be handled as follows:

```
% editor
editor: Command not found.
%
```

Your mistake in calling *edit editor* was treated by HP-UX as a request for a program named *editor*. Since there is no program named *editor*, HP-UX reported that the program or command could not be found. A new % prompt indicates that HP-UX is ready for another command, so you can now enter the correct command.

Summary

Your exchange with HP-UX as you logged in and made contact with *edit* should look something like this:

```
:login: susan
Password:
... A message of General Interest...
% edit text
"text" No such file or directory
:
```

Creating text

You can now begin to enter text into the buffer. This is done by appending text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing which, in effect, creates text. Most edit commands have two forms: a word that describes what the command does and a shorter abbreviation of that word. Either form can be used. Many beginners find the full command names easier to remember, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is *append* which can be abbreviated *a*. Type `APPEND`, then press `(Return)`.

```
% edit text
:aPPend
```

Messages from *edit*

If you make a mistake while entering a command and type something that *edit* does not recognize, *edit* responds with a message intended to help you diagnose your error. For example, if you misspell the command to input text by typing perhaps, *add* instead of *append* or *a*, you receive this message:

```
:add
add:Not an editor command
:
```

When you receive a diagnostic message, examine what you typed to determine what part of your command confused *edit*. The message above means that *edit* could not recognize your mistyped command, so the command was ignored. After displaying a new colon prompt, *edit* is now ready to receive a new command.

Text Input Mode

By giving the command *append* (or using the abbreviation *a*), you activated **text input mode**, also known as **append mode**. When you enter text input mode, *edit* responds by doing nothing. No prompts appear during text input mode, your signal to begin entering lines of text. You can type almost anything you want while inputting text lines. Lines are transmitted one at a time to the buffer and held there during the editing session. You can append as much text as you want. When you are through entering new text lines, type a period by itself at the beginning of a new line, then press `(Return)`. This signals the editor to terminate text input mode and return to **command mode**. *Edit* then prompts you for a new command by displaying a colon (:) prompt.

When you leave *append* mode and return to command mode (necessary in order to do any of the other kinds of editing, such as changing, adding, or printing text), *edit* preserves the text you just typed in the editor buffer, so nothing is lost. If you type any other character besides a period by itself on the last line, *edit* treats the line as text instead of an exit command, and will not let you leave *append*. To exit, type a period by itself on a single line terminated by `(Return)`.

This is a good place to learn an important lesson about computers and text: **as far as the computer is concerned, a blank space is a character as distinct as any letter of the alphabet. If you so much as type a blank after the period** (that is, type a period then press the space bar on the keyboard), **you will remain in append mode** with the last line of text being a period followed by a single space.

Let's say that the lines of text you enter are (try to type exactly what you see, including "this"):

```
This is some sample test.  
And this is some more text.  
Text editing is strange, but nice.
```

The last line is the period followed by **Return** that gets you out of append mode. If, while typing the line, you hit an incorrect character, you can change the incorrect character by using the **BACK SPACE** key to back up then retype the line beginning with the incorrect character. If you back-space to the first character in the line then press **Return**, a blank line is stored in the buffer. Corrections to a line must be done before the line has been completed by a **Return** (changes in lines already typed are discussed in Session 2).

Writing Text to Disc

Text input is now complete. Before you break for lunch, the text should be put in a disc file for safekeeping until the next editing session. Storing the editor's buffer in a disc file is the only way to save information from one session to the next, since the buffer is temporary and is destroyed after the end of the editing session. Thus, learning how to write a file to disc is second in importance only to entering the text. To write the contents of the buffer to a disc file, use the command, *write* (or its abbreviation *w*):

```
:write
```

Edit now copies the buffer to a disc file. If the file does not exist, a new file is created automatically and the presence of a "New File" will be noted. The newly-created file is given the name specified when you entered the editor, in this case, *text*. To confirm that the disc file has been successfully written, *edit* repeats the filename, then gives the number of lines and the total number of characters in the file. The buffer remains unchanged by the *write* command. All of the lines that were written to the disc are still in the buffer, should you want to modify or add to them.

This ability to write a file to the disc and still continue editing is useful insurance against loss of data during power failures. It is a good idea to periodically write the edit buffer to a permanent file to minimize the risk of losing an hour's work should the power go off for some reason (the risk is actually not as serious as this sounds, because HP-UX has recovery commands that recover all but very little of the file in the event of a power failure).

Edit must have a filename to use before it can write a file. If you forgot to indicate the name of the file when you began the editing session, *edit* prints:

```
No current filename
```

in response to your *write* command. If this happens, you can specify the filename in a new *write* command:

```
:write text
```

After the *write* (or *w*) type a space followed by the name of the file.

Logging Off

We have done enough for this first lesson on using the HP-UX text editor, and are ready to terminate (quit) the editing session. To do this, type *quit* (or *q*), then press **Return**. The terminal display looks like this:

```
:write
"text" [New file] 3 lines, 90 characters
"quit
%
```

The (%) prompt is from HP-UX, telling you that your session with *edit* is over and you can now interact with HP-UX. To end the entire session at the terminal, you must also exit from HP-UX. In response to the HP-UX prompt of “%” press the **CTRL** and **D** keys simultaneously to terminate the session with HP-UX and make the terminal available to the next user. It is always important to logout at the end of a session to make absolutely sure no one could accidentally stumble into your abandoned session and thus gain access to your files, a condition that tempts even the most honest of souls.

This is the end of the first session on HP-UX text editing.

Session 2

Login with HP-UX as in the first session:

```
:login:susan Return
Password: (give password then press Return)
%
```

This time when you type the *edit* command, you can specify the name of the file you worked on last time. Thus, when *edit* starts, it will transfer the original file into its buffer so that you can resume editing the same file. When *edit* has copied the file into the buffer, it shows the original file name, and lists the number of lines and characters in the file as follows:

```
%edit text
"text" 3 lines, 90 characters
:
```

Your command to edit file *text* caused the editor to copy the 90 characters of text into the buffer. *Edit* now awaits your next command. In this session you learn to append more text to the file, print the contents of the buffer, and change the text in a line.

Adding More Text to the File

To add more to the end of your text, use the *append* command to enter text input mode. When *append* is the first command of your editing session, the lines you enter are placed at the end of the buffer. Why this happens is explained later in this session. This time, use the abbreviation for the *append* command: *a*:

```
:a
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
:
```

Interrupt

Most terminals supported by HP-UX have a **DEL** (delete) key, sometimes labelled **RUN**. If you press **DEL** while working with *edit*, any task the editor is performing is stopped, and the following message is sent to you:

```
Interrupt
:
```

Any command that *edit* might be executing is terminated by **DELETE** or **RUN**, causing *edit* to prompt you for a new command. If you are appending text at the time the key is pressed, append mode terminates and you are expected to give another command. The line of text that you were typing when the *append* operation was interrupted is lost and is not entered into the buffer.

Making Corrections

If you have read a general introduction to HP-UX, such as *HP-UX User's Guide*, you will recall that it is possible to erase individual letters that you have typed. This is done by typing the designated erase character as many times as there are characters you want to erase. Accounts normally start out using the number sign (#) as the erase character, but it's possible for a different erase character to be selected. We'll show “#” as the erase character in our examples, but if you've changed your erase character to backspace (control-H) or something else, be sure to use your own erase character.

If you make a bad start in a line and would like to begin again, erasing individual characters with a “#” is cumbersome - what if you had 15 characters in your line and wanted to get rid of them? To do so either requires:

```
This is yukky tex*****
```

with no room for the great text you'd like to type, or,

```
This is yukky tex@This is great text.
```

When you type the at-sign (@), you erase the entire line typed so far. (An account can select a different line erase character to use in place of @. If your line-erase character has been changed, use it where the examples show “@”). You can immediately begin to retype the line. This, unfortunately, does not help after you type the line and press **Return**. To make corrections in completed lines, it is necessary to use the editing commands covered in this and following sessions.

HP-UX and *edit* also support use of **BACK SPACE** for text corrections. How the backspace key affects the terminal screen display depends on how your terminal or terminal emulator functions. You can look it up in the manual, or just try it out.

Listing Buffer Contents

Having appended text to what you wrote in Session 1, you might be curious to see what is in the buffer. To print the contents of the buffer, type the command:

```
:l,$P
```

The “1” stands for line 1 of the buffer, the “\$” is a special symbol designating the last line of the buffer, and *p* (for *print*) is the command to print from line 1 to the end of the buffer. Thus, `1,$p` gives you:

```
This is some sample text.  
And this is some more text.  
Text editing is strange, but nice.  
This is text added in Session 2. It doesn't mean much  
It doesn't mean much here, but  
it does illustrate the editor.
```

You may occasionally place a character in the buffer that cannot be printed (ASCII control characters are not printed on most output devices). These characters are usually obtained by pressing `CTRL` and some other key at the same time. When printing lines, *edit* uses a special notation to show the existence of non-printing (control) characters.

Suppose you had introduced the non-printing character “control-A” into the word “illustrate” by accidentally holding down the `CTRL` key while typing a. If you asked to have the line printed, `EDIT` would display:

```
it does illustr'Ate the editor.
```

The two-character sequence ‘A indicates that the CTRL key was depressed simultaneously with the “A” key, resulting in a corresponding control character (the apostrophe indicates that `CTRL` was pressed). The error is easily corrected, as discussed later in this session.

In looking over the text we see that “this” is typed as “thiss” in the second line, as was previously suggested. Let’s correct the spelling.

Finding Things in the Buffer

You must find something in the buffer before you can change it. To find “thiss” in the text you entered, look at a listing of the lines. *Edit* searches the buffer, looking for the text sequence “thiss”, and stops searching when it finds the specified character pattern. You can tell *edit* to search for a pattern by typing the pattern between slash marks:

```
:/thiss/
```

By typing `/thiss/` and pressing `Return`, *edit* is instructed to search for “thiss” (if *edit* cannot find the pattern of characters in the buffer, it responds “Pattern not found”). When *edit* finds the characters “thiss”, it prints the line where the pattern was found for your inspection:

```
And thiss is some more text.
```

Edit is now positioned in the buffer at the line which it just printed, ready to make a change in the line.

The Current Line

Edit always keeps track of its position in the buffer by identifying the “current line” at the end of each operation. In general, the line that was most recently printed, entered, or changed is considered to be the current position or line in the buffer. The editor assumes the next command is to be applied to the current line, unless you direct it to act in another location (or perform an operation that is not related to the current line). When you bring a file into the editor, the editor is always positioned at the last line in the file. If your initial editing command is “append”, the lines you enter are added to the end of the file, that is, they are placed after the current line. You can refer to your current position in the buffer by the symbol period (.), usually called “dot”. If you type “.” then press **(Return)**, you are telling *edit* to print the current line:

```
:.  
And thiss is some more text.
```

If you want to know the number of the current line, you can type number:

```
:.#  
2
```

If you type the number of any line and a carriage return, *edit* will position you at that line and print its contents:

```
:2  
And thiss is some more text.
```

Experiment with these commands to ensure that you understand what they do.

Numbering Lines (*nu*)

The *number* (*nu*) command is similar to *print*, giving both the number and the text of each printed line. To see the number and text of the current line, type

```
:nu  
2 And thiss is some more text.
```

Notice that the shortest abbreviation for the *number* command is *nu* (not *n* which is used for a different command). You can specify a range of lines to be listed by the number command in the same way that lines are specified for print. For example, `1, #nu` lists all lines in the buffer and their corresponding line numbers.

Substitute Command (*s*)

Now that you have found the misspelled word, it is time to change “thiss” to “this”. As far as *edit* is concerned, changing text is a matter of substituting one pattern for another. Just as *a* stood for *append*, so *s* stands for *substitute*. Use the abbreviation *s* to reduce the chance of mistyping the *substitute* command. This command instructs *edit* to make the change:

```
2s/thiss/this/
```

First, indicate the line to be changed (2), then type the command (*s*), followed by the characters to be removed (typed between slashes). Finish the line with the characters to be put back in followed by a closing slash mark, then press **(Return)**. Here it is in plain English:

```
2s/what is to be changed/what to change to/
```

If *edit* finds an exact match of the characters to be changed it makes the change **only** in the first occurrence of the characters. If it does not find the characters to be changed it will respond:

```
Substitute pattern match failed
```

indicating that your instructions could not be carried out. If *edit* finds the characters you want to change, it makes the substitution and automatically prints the changed line so you can verify that the correct substitution was made. In the example,

```
:2s/thiss/this/  
And this is some more text.  
:
```

line 2 (and line 2 only) is searched for the character pattern "thiss". When the first exact match is found, "thiss" is changed to "this". In reality, since you set the current line number to 2 in an earlier operation, it was unnecessary to specify the number of the line to be changed by this command. In the command:

```
:s/thiss/this/
```

edit assumes that the line where the editor is currently positioned (the **current line**) is to be used. A period can also be used to specify the current line as in the command:

```
:.s/thiss/this/
```

although the period is totally unnecessary. In either case, the command without a line number or without a period would have produced the same result as when the line number was specified because the editor was already positioned at the line to be changed. Here is another illustration of substitution.

```
Text editing is strange, but nice.
```

To be a bit more positive, take out the characters "strange, but" so the line reads:

```
Text editing is nice.
```

A command that positions *edit* at that line then makes the substitution is:

```
:/strange/s/strange, but //
```

This command combines the search with a substitution, a perfectly allowable combination. Thus, you do not necessarily have to use line numbers to identify a line to edit. Instead, you can identify the line to be changed by asking *edit* to search for a specified pattern of characters that occurs in the line of interest. The function of each part of the command is as follows:

```
/strange/      tells edit to find the characters "strange" in the text  
s              tells edit we want to make a substitution  
/strange,but // substitutes nothing at all for the characters "strange,  
but "
```

Note the space after "but" on "/strange, but /". If you do not indicate the space is to be taken out, your line becomes:

```
Text editing is nice.
```


which looks odd because of the extra space between “is” and “nice”. Again, you can see that a blank space is a real character to a computer, and when editing text you need to be aware of spaces within a line just as you would be aware of an “a” or a “4”.

Another Way to List What’s in the Buffer (z)

Although the *print* command is useful for looking at specific lines in the buffer, other commands can be more convenient for viewing large sections of text. You can ask to see a screen full of text at a time by using the command *z*. If you type

```
:1z Return
```

edit starts with line 1 and continues printing lines, stopping either when the screen of your terminal is full, or when the last line in the buffer has been printed. If you want to read the next segment of text, type the command

```
:z Return
```

If no starting line number is given for the *z* command, printing starts at the “current” line; in this case the last line printed. Viewing lines in the buffer one full screen at a time is known as paging. Paging can also be used to print a section of text on a printing terminal.

Saving the Modified Text

Now is a good place to pause and end the second session. If you hastily type **Return** to terminate the session, your interaction with *edit* resembles:

```
:q  
No write since last change (q! quits)  
:
```

This is *edit*'s warning that you have not written the modified contents of the buffer to disc. You are risking the loss of the work you have done during the editing session since the last previous *write* command. Since no previous disc write was performed during this session, everything done during the session would be lost. If you do not want to save the work done during this editing session, you can type *q!* to confirm that you indeed want to end the session immediately, losing the contents of the buffer. However, since you probably prefer to preserve the edited file, use the *write* command as follows:

```
:w  
"text" 6 lines, 171 characters
```

then follow with

```
:q  
%logout
```

and hang up the phone or turn off the terminal when HP-UX asks for a login name.

This is the end of the second session on HP-UX text editing.

Session 3

Bringing Text Into the Buffer (*e*)

Login to UNIX and make contact with *edit*. Try to do it without using notes if you can.

Did you remember to give the name of the file you wanted to edit by typing:

```
%edit text
```

or did you type:

```
%edit
```

Both commands activate *edit*, but only the first version can bring a copy of the file named *text* into the buffer. If you forgot to specify the filename, you can recover by typing:

```
:e text  
"text" 6 lines, 171 characters
```

The *edit* command which can be abbreviated *e* when you're in the editor, tells *edit* that you want to destroy anything already in the editor's buffer and copy the file *text* into the buffer for editing. You can also use the *edit* (*e*) command to change files in the middle of an editing session or to give *edit* the name of a new file that you want to create. Because the *edit* command clears the buffer, you will receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disc before editing the next file.

Moving Text in the Buffer (*m*)

Edit enables you to move lines of text from one location in the buffer to another by means of the *move* (*m*) command:

```
:2,4m$
```

This example directs *edit* to move lines 2, 3, and 4 to the end of the buffer following the last line, indicated by (\$). When constructing the *move* command, specify the first line to be moved, the last line to be moved, the move command *m*, then the line after which the moved text is to be placed. Thus,

```
:1,6m20
```

commands *edit* to move lines 1 through 6 (inclusive) to a position immediately following line 20 in the buffer. To move only one line, say line 4, to a position in the buffer after line 6, the command would be "4m6".

Let's move some text using the command:

```
:5,$m1  
2 lines moved  
it does illustrate the editor.
```

After executing a command that changes more than one line of the buffer, *edit* tells how many lines were affected by the change. The last moved line is printed for your inspection. If you want to see more than just the last line, use the print (p), z, or number (nu) command to view more text. The buffer should now contain:

```
This is some sample text,  
It doesn't mean much here, but  
it does illustrate the editor.  
And this is some more text.  
Text editing is nice.  
This is text added in Session 2.
```

You can restore the original order by typing:

```
:4,$m1
```

or you can combine context searching and the *move* command for the same result:

```
:/And this is some/,/This is text/m/This is some sample/
```

The danger in combining context searching with the move command lies in the higher probability of making a typing error in such a long command. Typing line numbers is usually much safer.

Copying Lines (*copy*)

The *copy* command is used to make a second copy of specified lines, leaving the original lines where they were. *Copy* has the same format as the *move* command. For example:

```
:12,15copy$
```

makes a copy of lines 12 through 15, placing the added lines after the last line in the buffer (\$). Experiment with the *copy* command so that you can become familiar with how it works. Note that the shortest abbreviation for *copy* is *co* (and not the letter *c* which has another meaning).

Deleting Lines (*d*)

Suppose you want to delete the line

```
This is text added in Session 2.
```

from the buffer. If you know the number of the line to be deleted, you can type that number followed by *delete* or *d*. This example deletes line 4:

```
:4d  
It doesn't mean much here, but
```

Here “4” is the number of the line to be deleted and “delete” or “d” is the command to delete the line. After executing the delete command, *edit* prints the resulting new current line (.).

If you do not happen to know the line, number you can search for the line then delete it using this sequence of commands:

```
:/added in Session 2./  
This is text added in Session 2.  
:d  
It doesn't mean much here, but
```

The “/added in Session 2./” asks *edit* to locate and print the next line containing the indicated text. Once you are sure that you have correctly specified the line you want to delete, you can enter the delete (d) command. In this case it is not necessary to specify a line number before the “d”. If no line number is given, *edit* deletes the current line (.), that is, the line found by the search operation. After the deletion, your buffer should contain:

```
This is some sample text.  
And this is some more text.  
Text editing is nice.  
It doesn't mean much here, but  
it does illustrate the editor.
```

To delete both lines 2 and 3:

```
And this is some more text.  
Text editing is nice.
```

type

```
:2,3d
```

to specify the range of lines (2 thru 3) and the operation on those lines (*d* for delete).

Again, this assumes that you know the line numbers for the lines to be deleted. If you do not, you can combine the *search* and *delete* commands as follows:

```
:/And this is some//Text editing is nice/d
```

This tells the editor to find the first line (following the current line) that contains the characters “And this is some”, then delete it and all subsequent lines until it has deleted the line containing “Text editing is nice”.

Be Careful

In using the search function to locate lines to be deleted, make absolutely sure that the characters you give as the basis for the search will take *edit* to the line you want deleted. *Edit* searches for the first occurrence of the characters starting from where you last edited: that is, from the line you see printed if you type a period (.) then press **Return**.

A search based on too few characters may result in the wrong line being deleted (if an identical pattern appears elsewhere in the text). For this reason, it is usually safer to specify the search, then delete in a second separate step, at least until you become familiar enough with the editor that you understand how best to specify searches. For beginners, be safe and double-check each command before pressing **Return** to send the command on its way.

Oops! I goofed. Now what? (*undo*)

The *undo* (*u*) command has the ability to reverse the effects of the last (and only the last) command. To undo the previous command type *u* or *undo*. *Undo* can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give. *Undo* affects only those commands that can change the buffer, such as *delete*, *append*, *move*, *copy*, *substitute*, and even *undo* itself. The commands *write* (*w*) and *edit* (*e*) which interact with disk files cannot be undone, nor can commands such as *print* which do not change the buffer. Most important: the only command that can be reversed by *undo* is the last “undo-able” command preceding the *undo*.

To illustrate, let’s issue an *undo* command. Recall that the last buffer-changing command deleted the lines that were formerly numbered 2 and 3. Executing *undo* at this time reverses the effects of the deletion, causing those two lines to be restored to their original position in the buffer.

```
:u
2 more lines in file after undo
And this is some more text.
```

Again, as before, *edit* informs you when the command affects more than one line, and prints the text of the resulting new current line.

More About Dot (.) and Buffer End (\$)

The function assumed by the dot symbol (period) depends on its context. It can be used to:

- Exit from append mode by typing a period (by itself) followed immediately by Return
- Refer to the current line in the editor’s buffer.

A period can also be combined with an equal sign to get the number of the line currently being edited (current line):

```
:.=
```

Thus, type *.=* to ask for the number of the current line, or use a colon instead of the equal sign (*.:*) to ask for the text in the current line.

In this editing session, as in the last, the dollar sign was used to indicate the last line in the buffer for commands such as *print*, *copy*, and *move*. As a command, the dollar sign asks *edit* to print the last line in the buffer. If the dollar sign is combined with the equal sign (*\$=*), *edit* prints the line number corresponding to the last line in the buffer.

(*.*) and (*\$*) therefore represent line numbers. Whenever appropriate, these symbols can be used in place of line numbers in commands. For example:

```
:.,$d
```

instructs *edit* to delete all lines from the current line (*.*) through the last line in the buffer.

Moving Around in the Buffer (+ and -)

It is frequently convenient during an editing session to go back and re-read a previous line. You could specify a context search for a line you want to read if you remember some of its text, but if you simply want to see what was written a few (say, 3) lines ago, you can type:

```
-3P
```

This tells *edit* to move back to a position 3 lines before the current line (.) and print that line. You can move forward in the buffer similarly:

```
+2P
```

tells *edit* to print the line which is 2 ahead of our current position. You can use + and - in any command where *edit* accepts line numbers. Line numbers specified with "+" or "-" can be combined to print a range of lines. The command:

```
:-1,+2copy$
```

copies 4 lines: the line preceding the current line, the current line, and the two lines following the current line, placing them after the last line in the buffer (\$).

Try typing a single minus (-). You will move back one line just as if you had typed, :-1P. Typing the command "+" works similarly. You might also try typing a few plus or minus signs in a row (such as "+ + +") to see *edit*'s response. Typing a carriage return alone on a line is the equivalent of typing "+1p": it moves you one line ahead in the buffer and prints that line.

If you are at the last line in the buffer and try to move further ahead, perhaps by typing a "+" or a carriage return alone on the line, *edit* reminds you that you are at the end of the buffer:

```
At end-of-file
```

Similarly, if you try to move to a position before the first line, *edit* will print one of these messages:

```
Nonzero address required on this command  
Negative address - first buffer line is 1
```

The number associated with a buffer line is the line's "address", in that it can be used to locate the line.

Changing Lines (c)

There may be occasions when you want to delete certain lines and insert new text in their place. This can be accomplished easily with the *change* (c) command. The change command instructs *edit* to delete specified lines then switch to text input mode in order to accept the text that will replace them. Let's assume that you want to change the first two lines in the buffer:

```
This is some sample text.  
And this is some more text.
```

to read

```
This text was created with the HP-UX text editor.
```

To do so, you can type:

```
:.,2c
2 lines changed
This text was created with the HP-UX text editor.
.
:
```

The command *1,2c*, specifies that you want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the *print* command. These lines will be deleted. After a u86 enters the change command, *edit* notifies you if more than one line is being changed, then places you in text input mode. Any text typed on the following lines is inserted into the position where lines were deleted by the change command. You remain in text input mode until you exit by typing a period alone on a line. Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

This is the end of the third session on text editing with HP-UX.

Session 4

This lesson covers several topics, starting with commands that affect the entire buffer, characters with special meanings, and how to issue HP-UX commands while using the editor. The next topics deal with files, discussing more about reading and writing, and explaining how to recover files lost in a crash. The final section provides leads to other sources of information and other editors that expand beyond *edit*.

Making Commands Global (*g*)

One disadvantage of using the commands in the manner illustrated when searching or substituting is that if you have a number of instances of a word to change, it would appear that you have to type the command repeatedly, once for each time the change needs to be made. *Edit*, however, provides a way to make commands apply to the entire contents of the buffer – the *global (g)* command. To print all lines containing a certain sequence of characters (say, “text”) the command is:

```
:g/text/p
```

The *g* instructs *edit* to make a global search for all lines in the buffer containing the characters *text*. The *p* prints the lines found.

To issue a global command, start by typing a “g” and then a search pattern identifying the lines to be affected. Then, on the same line, type the command to be executed on the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word “text” to the word “material” the command would be a combination of the global search and the substitute command:

```
:g,text/s/text/material/g
```

Note the “g” at the end of the global command which instructs edit to change each and every instance of “text” to “material”. If you do not type the “g” at the end of the command, only the first instance of “text” in each line will be changed (the normal result of the substitute command). The “g” at the end of the command is independent of the “g” at the beginning. You can give a command such as:

```
:l4x/text/material/g
```

to change every instance of “text” in line l4 alone. Note further that neither command will change “Text” to “material” because “Text” begins with a capital rather than a lower-case t. Edit does not automatically print the lines modified by a global command. If you want the lines to be printed, type a “p” at the end of the global command:

```
:g/text/s/text/material/gp
```

The usual qualification should be made about using the global command in combination with any other. Be sure you know what you are telling *edit* to do to the entire buffer. For example:

```
:g/ /d  
72 less lines in file after global
```

deletes every line containing a blank anywhere in it. This could demolish your document, because most lines contain spaces between words, and thus would be deleted. After executing the global command, *edit* prints a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. Try experimenting with the global command on a small buffer of text to see what it can do for you.

More about Searching and Substituting

Previous examples of using slashes to identify a character string that you want to search for or change have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change “noun” to “nouns” you can type either

```
:/noun/s/noun/nouns/
```

as before, or use a somewhat abbreviated command:

```
"/noun/s//nouns/
```

In this example, the characters to be changed are not specified (there are no characters, not even a space, between the two slash marks that indicate what is to be changed). This lack of characters between the slashes is taken by the editor to mean “use the characters we last searched for as the characters to be changed”.

Similarly, the last context search can be repeated by typing a pair of slashes with nothing between them:

```
"/does/  
It doesn't mean much here, but  
://  
it does illustrate the editor
```


Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters “does”.

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the character string.

It is also possible to repeat the last substitution without having to retype the entire command. An ampersand (&) used as a command repeats the most recent substitute command, using the same search and replacement patterns. After altering the current line by typing

```
:s/noun/nouns/
```

you could use the command

```
:/nouns/&
```

or simply

```
://&
```

to make the same change on the next line in the buffer containing the characters “nouns”.

Special Characters

Two characters have special meanings when used in specifying searches: the dollar sign (\$), and circumflex (^). (\$) is taken by the editor to mean “end of the line” and is used to identify strings which occur at the end of a line.

```
:s/ing&/s//ed/p
```

tells the editor to search for all lines ending in “ing” (and nothing else, not even a blank space) to change each final “ing” to “ed” and print the changed lines.

The circumflex (^) indicates the beginning of a line. Thus,

```
:s/^/1. /
```

instructs the editor to insert “1.” and a space at the beginning of the current line.

These characters, (\$) and (^), have special meanings only in the context of searching. At other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to temporarily lose its special significance by typing another special character, the backslash (\), before it.

```
:s/\$/dollar/
```

looks for the character “\$” in the current line and replaces it by the word “dollar”. Were it not for the backslash, the “\$” would have represented “the end of the line” in your search, rather than the character “\$”. The backslash retains its special significance unless it is preceded by another backslash.

Issuing HP-UX Commands from the Editor

After creating several files with the editor, you may want to delete files no longer useful to you or ask for a list of your files. Removing and listing files are not editor functions, so they require use of HP-UX system commands (also referred to as "shell" commands, because the HP-UX program that processes HP-UX commands is called a "shell"). You do not need to quit the editor to execute an HP-UX command as long as you indicate that it is to be sent to the shell for execution. To use the HP-UX command *rm* to remove the file named *junk*, type:

```
:!rm junk
!  
:
```

The exclamation point (!) indicates that the rest of the line is to be processed as an HP-UX command. If the buffer contents have not been written since the last change, a warning is printed before the command is executed. The editor replies with an exclamation point when the command is completed. The *Getting Started with HP-UX* manual describes useful features of the system, and is helpful background when you need to access HP-UX from *edit*.

Filenames and File Manipulation

Throughout each editing session, *edit* keeps track of the name of the file being edited as the current filename (the current filename is the name given when you entered the editor). The current filename changes whenever the edit (*e*) command is used to specify a new file. Once *edit* has recorded a current filename, it inserts that name into any command where a filename has been omitted. If a write command does not specify a file, *edit*, as you have seen, supplies the current filename. You can have the editor write all or part of its buffer contents to a different file by including the new file name in the *write* command:

```
:wchapter3  
"chapter3" 283 lines, 8698 characters
```

The current filename remembered by the editor does not change as a result of the *write* command unless it is the first filename given in the editing session. Thus, using the previous example, the next *write* command that does not specify a file name will write onto the current file, not onto the file *chapter3*.

The File (*f*) Command

To ask for the current filename, type *file* (or *f*). In response, the editor provides updated information about the buffer, including the filename, your current position, and the number of lines in the buffer:

```
:f  
"text" [Modified] line 3 of 4--75%--
```

If the contents of the buffer have changed since the last time the file was written, the editor will tell you that the file has been "Modified". After you save the changes by writing to a disc file, the buffer is no longer considered modified:

```
:w  
"text"4 lines, 88 characters  
:f  
"text"line 3 of 4--75%--
```

Reading Additional Files (*r*)

The read (*r*) command enables you to add the contents of a file to the buffer without destroying the text already there. To use it, specify the line after which the new text is to be placed, the command *r*, then the name of the file.

```
:$r bibliography
"bibliography" 18 lines, 473 characters
```

This command reads in the file *bibliography* and adds it to the buffer after the last line. The current filename is not changed by the *read* command unless it is the first filename given in the editing session.

Writing Parts of the Buffer

The *write* (*w*) command can write all or part of the buffer to any file you specify. You are already familiar with writing the entire contents of the buffer to a disc file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command. For example:

```
:45,$w ending
```

Here all lines from 45 through the end of the buffer are written to the file named *ending*. The lines remain in the buffer as part of the document you are editing, so you can continue to edit the entire buffer.

Recovering Files

Under most circumstances, *edit*'s crash recovery mechanism is able to save work to within a few lines of changes after a crash or if your terminal is accidentally disconnected. If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login, listing the name of the recovered file. To recover the file, enter the editor and type the command *recover* (*rec*), followed by the name of the lost file.

```
:recover chap6
```

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

Other Recovery Techniques

If something goes wrong while you are using the editor, it may be possible to save your work by using the command *preserve* (*pre*), which saves the buffer as if the system had crashed. If you are writing a file and receive the message "Quota exceeded", you have tried to use more disc storage than is allotted to your account. Proceed with caution because it is likely that only a part of the editor's buffer is now present in the file you tried to write. In this case, you should use the shell escape from the editor (!) to remove some files you don't need and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

```
:PRESERVE
```

then seek help. Do not simply leave the editor. If you do, the buffer will be released (and possibly destroyed), and you may not be able to save your file. After a *preserve*, you can use the *recover* command once the problem has been corrected.

If you make an unwanted change to the buffer and issue a *write* command before discovering your mistake, the modified version will replace any previous version of the file. Should you ever lose a good version of a document in this way, do not panic and leave the editor. As long as you stay in the editor, the contents of the buffer remain accessible. Depending on the nature of the problem, it may be possible to restore the buffer to a more complete state with the *undo* command. After fixing the damaged buffer, you can again write the file to disc.

Further Reading and Information

Edit is an editor designed for beginning and casual users. It is actually a version of a more powerful editor called *ex*. These lessons are intended to introduce you to the editor and its most commonly used commands. We have not covered all of the editor's commands, just a selection of commands which should be sufficient to accomplish most of your editing tasks. You can find out more about the editor in the *ex* tutorial, which is applicable to both *ex* and *edit*. One way to become familiar with *ex* is to begin by reading the description of commands that you already know.

Using *ex*

As you become more experienced with using the editor, you may still find that *edit* continues to meet your needs. However, should you become interested in using *ex*, it is easy to switch. To begin an editing session with *ex*, use *ex* in your command instead of *edit*.

Edit commands work the same way in *ex*, but the editing environment is somewhat different. You should be aware of a few differences that exist between the two versions of the editor. In *edit*, only the characters `^`, `$`, and `\` have special meanings in searching the buffer or indicating characters to be changed by a *substitute* command. Several additional characters have special meanings in *ex*, as described in the *ex* tutorial. Another feature of the *edit* environment prevents users from accidentally entering two alternative modes of editing, **open** and **visual**, in which the editor behaves quite differently than in normal command mode. If you are using *ex* and the editor behaves strangely, you may have accidentally entered *open* mode by typing *o*. Type the ESC key and then a "Q" to get out of open or visual mode and back into the regular editor command mode. *The Vi Editor* provides a full discussion of visual mode.

Table of Contents

Ex Extended Editor

Starting ex.	1
File Manipulation	2
Current File	2
Alternate File	2
Filename Expansion	2
Multiple Files and Named Buffers	2
Read-only Operation	3
Exceptional Conditions	3
Errors and Interrupts	3
Recovering from Hangups and Crashes	3
Editing Modes	4
Command Structure	4
Command Parameters	4
Command Variants	5
Flags After Commands	5
Comments	5
Multiple Commands per Line	5
Reporting Large Changes	5
Command Addressing	6
Addressing Primitives	6
Combining Addressing Primitives	6
Command Descriptions	7
Regular Expressions and Substitute Replacement Patterns	16
Regular Expressions	16
Magic and Nomagic	16
Summary of Basic Regular Expressions	16
Combining Regular Expression Primitives	17
Substitute Replacement Patterns	17
Options Descriptions	18
Limitations	22
Ex Changes - Version 3.1 to 3.5	22
Update to Ex Reference Manual	22
Command Line Options	22
Commands	22
Options	22
Environment Enquiries	23
Vi Tutorial Update	23
Deleted Features	23
Change in Default Option Settings	23
Vi Commands	24
Macros	24



Ex Extended Editor

The *ex* editor has many options that can be set to meet individual needs. It is much more comprehensive and more versatile than the *edit* version that uses predefined defaults for some options to better fit the needs of beginning and casual users. In this tutorial, default settings are assumed for all command options unless stated otherwise.

Starting *ex*

When invoked, *ex* uses the environment variable *TERM* to determine the terminal type. If a *TERMCAP* variable in the environment matches the terminal described by the *TERM* variable, that description is used. Also if the *TERMCAP* variable contains a pathname (beginning with */*), the editor seeks the description of the terminal in that file (rather than the default */etc/termcap*). If there is a variable *EXINIT* in the environment, the editor executes the commands contained in that variable. Otherwise, if there is a file *.exrc* in your HOME directory, *ex* reads commands from that file, thus simulating a source command. Option-setting commands placed in *EXINIT* or *.exrc* are executed before each editor session.

The *ex* start-up command has the following prototype:

```
ex [-] [-v] [-t <tag>] [-r][-l] [-w<n>] [-x] [-R] [+ <command>] <name> . . .
```

where brackets ([]) surround optional command parameters. The most common case edits a single file with no options, i.e.:

```
ex name
```

Command-line options function as follows:

- Suppresses all interactive-user feedback; useful when processing editor scripts in command files.
- v Equivalent to using *vi* rather than *ex*.
- t Equivalent to an initial tag command. Edits the file containing the tag and positions the editor at its definition.
- r Used in recovering after an editor or system crash. retrieves the last saved version of the named file or, if no file is specified, types a list of saved files.
- l Sets up for editing LISP, by setting the showmatch and lisp options.
- w Sets the default window size to *n*, and is useful on dial-ups to start in small windows.
- x Causes *ex* to prompt for a key that is then used to encrypt and decrypt the contents of the file. The file should have been previously encrypted using the same key, see *crypt(1)*.
- R Sets the read-only option at the start.
- name* Indicates which file(s) to edit.

An argument of the form + <command> indicates that the editor should begin by executing the specified command. If <command> is omitted, the argument defaults to "S", initially positioning the editor at the last line of the first file. Other useful commands here are scanning patterns of the form /*pattern*, or line numbers such as +100 (which starts at line 100).

File Manipulation

Current File

In normal use, *ex* is used to edit the contents of a single file whose name is specified by the **current** filename. In a typical editing sequence, the name of the file to be edited becomes the **current** filename, and the original file contents are copied into a buffer which is actually a temporary buffer file. *Ex* performs all editing actions on the buffer file. Changes made to the buffer have no effect on the file being edited unless and until the original file is replaced by the edited buffer contents (by use of a write command). The write operation destroys the original file and replaces it with the edited version.

The current file is almost always treated as having been edited. This means that the buffer file contents are logically connected with the current file name so that writing the current buffer contents onto that file, even if it exists, is a reasonable action. If the original file has not been edited, then *ex* will not normally write on it if it already exists (a "not edited" message is returned when the write operation is attempted).

Alternate File

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

Filename Expansion

Filenames within the editor can be specified using normal shell-expansion conventions. In addition, the character % in filenames is replaced by the current file name; the character # is replaced by the alternate file name (this makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an edit command after a No write since last change diagnostic is received).

Multiple Files and Named Buffers

If the command line specifies more than one file to be edited, the first file is edited as previously explained. Command-line arguments and file names for the first and subsequent files to be edited are placed in the argument list (the current argument list can be displayed by using the *args* command). When you are ready to edit the next file in the list, use the *next* command. If you want to destroy the original argument list and associated file names, replacing them with a new list, append the desired new arguments and file names to the *next* command. HP-UX then expands the *next* command with its new arguments. The resulting list of names becomes the new argument list; the old list is destroyed, and *ex* edits the first file on the new list.

Ex has a group of named buffers that are particularly useful for saving blocks of text during normal editing, especially when editing multiple files. These buffers are similar to the normal buffer file, except that only a limited number of operations can be used with them. The buffers have names *a* or *A* through *z* or *Z*. Uppercase and lowercase names refer to the same buffers, but commands *append* to uppercase-named buffers and *replace* lowercase-named buffers.

Read-only Operation

You can use *ex* in read-only mode to look at files that you have no intention of modifying, thus preventing the possibility of accidentally overwriting a file. Read-only mode is active when the **readonly** option is set by:

- Using the `-R` command-line option,
- The `view` command line invocation, or
- By setting the `readonly` option.

Read-only can be cleared by setting `noreadonly` (type: `: noreadonly RETURN`). It is possible to write, even while in read-only mode, by indicating that you really know what you are doing. You can write to a different file or use the `!` form of write, even while in read-only mode.

Exceptional Conditions

Errors and Interrupts

When errors occur, *ex* prints an error diagnostic and, optionally, rings the terminal bell. If the primary input is from a file, editor processing terminates. If an interrupt signal is received, *ex* prints “Interrupt” and returns to its command level. If the primary input is a file, *ex* exits when an interrupt occurs.

Recovering from Hangups and Crashes

If a hangup signal is received and the buffer has been modified since it was last written out, or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second case) attempts to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing, at most, a few lines of changes from the last point before the hangup or editor crash. To recover a file, you can use the `-r` option. For example, if you were editing the file *resume*, you should change to the directory you were using when the crash occurred, giving the command

```
ex -r resume
```

After checking that the retrieved file is indeed intact, you can write it back over the original unedited file. The system normally sends you mail, telling you when a file has been saved after a crash. The command

```
ex -r
```

prints a list of the files that have been saved for you. (In the case of a hangup, the file does not appear in the list, although it can be recovered.)

Editing Modes

Ex has five distinct operating modes:

- **Command mode** where commands are entered when a colon (:) prompt is present and executed each time a complete line is sent.
- **Text-input mode** where *ex* gathers incoming lines of text and places them in the file. Append, insert, and change commands use text-input mode to alter existing text.
No prompt is printed when you are in text-input mode. To exit this mode, type a period (.) immediately followed by an end-of-line key (**RETURN**). Command mode then resumes.
- **Open and visual** modes enable you to perform local editing operations on text in the file. The modes are accessed by commands having the same name. The *open* command displays text, one line at a time, on any terminal, while the *visual* command is designed for CRT terminals that have direct screen-cursor addressing capability so *ex* can use the CRT as a window for file-editing changes.
- **Text insertion** mode operates within *open* and *visual* modes.

These modes are discussed in greater detail in *The Vi Editor*.

Command Structure

Most command names are English words, and initial prefixes of the words are acceptable abbreviations. Ambiguous abbreviations are resolved in favor of the more commonly used commands (for example, the command *substitute* can be abbreviated *s*, while the shortest available abbreviation for *set* is *se*).

Command Parameters

Most commands accept prefix addresses specifying which line(s) they are to affect. The forms these addresses can take is discussed below. Some commands also accept or require a trailing count specifying the number of lines to be affected by the command (if rounding is necessary, the number is rounded down). Thus the command `10P` prints the tenth line in the buffer while `delete 5` deletes five lines from the buffer, starting with the current line.

Some commands require other information or parameters that are always appended following the command name; for example, option names in a *set* command, a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command as in `1,5 copy 25`.

Command Variants

A number of commands have two distinct variants. The variant form of the command is invoked by placing an exclamation point (!) immediately after the command name. Some of the default variants can be controlled by options; in this case, the ! serves to toggle the default.

Flags After Commands

The characters #, #, and l can be placed after many commands (a p or l must be preceded by a blank or tab except in the single special case dp). The commands abbreviated by these three characters are executed after the command completes. Since ex normally prints the new current line after each change, p is rarely necessary. Any number of + or - characters can also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

Comments

Comment commands are ignored by the editor. This feature is useful when making complex editor scripts where explanatory comments are needed. Any line beginning with a double quotation mark (") is treated as a comment and no action results. Comments beginning with " can also be placed at the ends of commands, except in cases where they could be confused as part of text (as in shell escape sequences or in substitute or map commands).

Multiple Commands per Line

Multiple commands can be combined on a single line by separating adjacent commands with a ; character. However, global commands, comments, and the shell escape ! must be the last command on a line because they are not terminated by a ;.

Reporting Large Changes

Most commands that change the editor buffer contents give feedback whenever the scope of the change exceeds a threshold set by the report option. This feedback helps detect undesirably large changes so that they can be quickly and easily reversed with an *undo*. When using commands that have a more global effect (such as *global* or *visual*) you will be informed if the net change in the number of lines in the buffer during this command exceeds the threshold.

Command Addressing

Addressing Primitives

- `.` (period) The current line, usually the last line affected by the previous command. The default address for most commands is the current line, thus `.` is rarely used alone as an address.
- `n` The *n*th line in the buffer file, lines being numbered sequentially from 1.
- `$` The last line in the buffer.
- `%` An abbreviation for `"1,$"`, which addresses the entire buffer.
- `+n -n` An offset relative to the current buffer line. The forms `+3`, `.,+3`, and `+++` are all equivalent. If the current line is 100, they all address line 103.
- `/<pattern>/`
or
`?<pattern>?` Search forward (/) or backward (?) respectively for a line containing `<pattern>` where `<pattern>` is any regular expression, usually a string of text characters. Searches normally wrap around the end of the buffer. If you only want to print the next line containing `<pattern>`, the trailing / or ? can be omitted. If `<pattern>` is omitted or explicitly empty, the last previous regular expression used in a pattern search is substituted for `<pattern>`.
- `" ' <x>` Used to locate previously-marked lines. Before each non-relative motion of the current line (`.`), the previous current line is marked with a tag, subsequently referred to as `"`. This makes it easy to refer or return to this previous context. Marks can also be established by the *mark* command, using single lowercase letters `<x>`. Marked lines are then referred to by `'<x>`.

Combining Addressing Primitives

Addresses to commands consist of a series of addressing primitives, separated by `(,)` or `(;)`. Such address lists are evaluated left-to-right. When addresses are separated by `(;)` the current line (`.`) is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, all but the last one or two are ignored. If the command requires two addresses, the first addressed line must precede the second in the buffer. Null address specifications are permitted in a list of addresses. The default in this case is the current line, so `.,100` is equivalent to `.,100`. Giving a prefix address to a command that expects none produces an error diagnostic.

Command Descriptions

All *ex* commands have the following form:

<address> <command> ! <parameters> <count> <flags>

All parts are optional; the degenerate case is the empty command which prints the next line in the file. To preserve user sanity when operating in visual mode, *ex* ignores a `:` preceding any command.

In the following command descriptions, the command, its standard abbreviation, and default value (if any) are shown in the left column.

abbreviate <word> <text>
abbr: *ab*

Add the specified abbreviation to the current list. <Word> is the abbreviated form of <text> that is being defined by the command. When using *visual* in input mode, if <word> is typed as a complete word (blanks before and after), *ex* expands the abbreviation, and displays <text> (the expanded form is also used in the buffer).

append <text>
abbr: *a* <text>
Default: current line

Reads input <text> and places it after the specified line. If preceded by period (.) it addresses the last line input or the specified line if no lines were input. If address zero (0) is given, <text> is placed at the beginning of the buffer.

a! <text>
Default: current line

The variant flag on *append* toggles the setting for the auto-indent option during text input.

args

The members of the argument list are printed, with the current argument delimited by left and right brackets ([]).

change <count> <text>
abbr: *c*
Default: current line

Replaces lines specified by <count> with the input <text>. Upon completion, the current line becomes the last line in <text>. If no text is provided, the command is treated as a *delete*.

c! <text>
Default: current line

The variant flag on *change* toggles the setting for the auto-indent option during text input.

copy <address> <flags>
abbr: *co*
Default: current line

A copy of the lines specified by <flags> is placed after <address> which can be zero. Upon completion, the current line (.) addresses the last line of the copy. The *t* command is a synonym for *copy*.

delete <buffer> <count> <flags>
abbr: *d*
Default: current line

Removes the lines specified by <count> and <flags> from the text buffer file, and the line following the last line deleted becomes the new current line. If the deleted lines were originally at the end of the text buffer file, the new last line in the file becomes the current line. If a named <buffer> is specified by a single letter, the deleted lines are saved in that buffer. If the buffer name is lowercase, previous buffer contents are overwritten; if uppercase, the lines are appended to any existing text in the buffer.

edit <file>
abbr: *e* or
ex <file>
(no abbr)

Used to begin an editing session on new file(s). The editor first checks to see if the buffer has been modified since the last write command was issued. If so, a warning is issued and the command is aborted. Otherwise, the command clears the entire editor buffer, makes the named file the current file, and prints the new filename.

After ensuring that this file looks reasonable, the editor reads the source file into its buffer. If the transfer is completed without error, the number of lines and characters in the file is typed. If there were any non-ASCII characters in the file, their non-ASCII high bits are stripped, and any null characters are discarded.

If none of these errors occurred, the file is still considered edited. If the last line of the input file is missing the trailing newline character, it will be supplied and a complaint will be issued. This command leaves the current line (.) at the last line read.

e! <file>

The variant flag on *edit* forces the specified **new** <file> to be transferred to the editor buffer for editing whether or not the current file has been preserved with a write command, and suppresses all complaint messages caused by the forced command.

e + <n> <file>

Causes the editor to begin at line <n> rather than at the last line; <n> can also be an editor command containing no spaces such as */<pattern>*.

file

Prints the current file name and provides the following information: whether the file has been modified since the last write command; whether it is read-only; current line number; number of lines in the buffer; and the relative location of the current line in the buffer (expressed as a percentage).

file <file>

The current filename is changed to <file> which is considered **not edited**.

global/*<pattern>*/*<commands>*
abbr: *g*
Default: all lines

First marks each line among those specified that matches the regular expression in <pattern>. Given <commands> are then executed with the current line initially set to each marked line.

The command list consists of the remaining commands on the current input line and can continue to multiple lines by ending all but the last such line with a slash (/). If `<commands>` (and possibly the trailing (/) delimiter) is omitted, each line matching `<pattern>` is printed. Append, insert, and change commands and their associated input are permitted. The (.) terminator normally required for insert, append, and change can be omitted if the end of the `<text>` associated with the command coincides with the end of the *global* command. *Open* and *visual* commands are permitted in the command list and take input from the terminal.

The *global* command itself must not appear in `<commands>`. The *undo* command is also not permitted there, because *undo* could be used to reverse the entire global command. The options *autoprint* and *autoindent* are inhibited during a global (and possibly the trailing / delimiter). The value of the *report* option is temporarily infinite, in deference to a report for the entire global. Finally, the context mark (") is set to the value of (.) before the global command begins, and is not changed during a global command except, perhaps, by an *open* or *visual* within the global.

`g!<pattern>/<commands>`
abbr: *v*

`insert <text>`
abbr: *i*
Default: current line

`!<text>`
Default: current line

`join <count> <flags>`
abbr: *j*
Default: current and next line

The variant form of *global* runs `<commands>` at each line not matching `<pattern>`.

Places `<text>` before the specified line. Upon completion, the new current line becomes the last line in `<text>`. If no `<text>` is given, it is set to the line before the addressed line. This command differs from *append* only in the placement of text. `<Text>` must be terminated by a period alone on a single line (except for certain cases in *global*).

The variant form of *insert* toggles *autoindent* during the *insert*.

Places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character, two if there was a comma at the end of the line, or none if the first following character is a right-hand parenthesis. If there is already white space at the end of the line, the white space at the start of the next line is discarded.

j!

The variant of *join* causes a simpler join with no white-space processing. Characters in the lines are simply concatenated, and no spaces are eliminated.

k<x>

The *k* command is a synonym for *mark*. It does not require a blank or tab before the following letter <x>.

list <count> <flags>

Prints the specified lines in a more unambiguous way: tabs are printed as “I” and the end of each line is marked with a trailing “\$”. The current line is left at the last line printed.

map <key> <replacement>

The *map* command is used to define macros for use in *visual* mode. <Key> should be a single character, or the sequence “#n” where <n> is a digit referring to function key <n>. When the character or key specified by <key> is typed in *visual* mode, the corresponding <replacement> expression is substituted (and displayed). On terminals without function keys, you can type “#n”. See *The Vi Editor* for more details.

mark <x>

Default: current line

Places the specified mark <x> on the current line. <x> is a single lowercase letter that must be preceded by a blank or a tab. The marking character <x> can then be used in subsequent commands to address this line. The current line does not change.

move <address>

abbr: *m*

Default: current line only

The *move* command repositions the specified lines to be after <address>. The first of the moved lines becomes the current line.

next

abbr: *n*

Starts editing next <file> in argument list.

n!

The variant suppresses warnings about the modifications to the editor buffer not having been written out, and irretrievably discards any changes that may have been made.

n <filelist>

or

n + <command> <filelist>

The specified filelist is expanded and the resulting list replaces the current argument list. The first file in the new list is then edited. If <command> is given (it must contain no spaces), it is executed after editing the first file in the new list.

number <count> <flags>

abbr: # or *nu*

Default: current line

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

open <flags>

or

open /<pattern> /<flags>

Default: current line

Enters intraline editing open mode at each addressed line. If a pattern is given, then the cursor will be placed initially at the beginning of the string matched by the pattern. To exit this mode use *Q*. See *The Vi Editor* for more details.

preserve

The current editor buffer is saved just as it would be in a system crash. This command is for emergency use when a write command has resulted in an error and you don't know how to save your work. After a *preserve* you should seek help.

print <count>

abbr: *p* or *P*

Default: current line only

Prints the specified lines with non-printing characters printed as control characters '^x'; delete (octal 177 is represented as '^?'). The current line is left at the last line printed.

put <buffer>

abbr: *pu*

Default: current line

Puts back previously deleted or yanked lines. Normally used with *delete* to move lines, or with *yank* to duplicate lines. If no buffer is specified, the last deleted or yanked text is restored. By using a named buffer, text can be restored that was saved there at any previous time.

quit

abbr: *q*

Causes *ex* to terminate. No automatic write of the editor buffer to a file is performed, but *ex* issues a warning message if the file has changed since the last write command was issued, and aborts the quit. If you want to save your changes before leaving, give a write command. To discard the changes, use the *q!* command variant to force termination.

q!

Forced *quit* from the editor. Discards changes to the buffer without complaint.

read <file>

abbr: *r*

Default: current line

Copies text from <file> into the editing buffer beginning after the specified line. If no <file> is given, the current file name is used. If <file> is specified, it does not change the current file name unless no current file name exists (in which case <file> becomes the current filename). Sensibility restrictions for the *edit* command also apply here. If the file buffer is empty and there is no current filename, *ex* treats *read* as an *edit* command.

Address zero is legal for this command, and causes the file to be read at the beginning of the buffer. Statistics are given as for the *edit* command when the *read* successfully terminates. After a *read*, the last line read becomes the current line.

read !<command>

Default: current line

Places the output of <command> in the editor buffer after the specified line. This is not a variant form of *read*, but rather a *read* specifying a <command> rather than a <filename>. A blank or tab before the (!) is mandatory.

recover <file>

Recovers <file> from the system save area. Used after an accidental hangup of the phone, a system crash, or after a *preserve* command. You will be notified by mail when a file is saved (except after a *preserve*).

rewind
abbr: *rew*

The argument list is rewound, and the first file in the list is edited.

rew!

Rewinds the argument list, discarding any changes made to the current buffer.

set <parameter>

If <parameter> is not given, only those options whose values have been changed from their defaults are printed. If <parameter> is given, all option values are printed.

Giving an option name followed by a '?' causes the current value of that option to be printed. The '?' is unnecessary unless the option is Boolean-valued. Boolean options are given values either by the form 'set option' to turn them on or 'set nooption' to turn them off; string and numeric options are assigned via the form 'set option = value'. More than one parameter may be set in a single invocation of *set*; they are interpreted left-to-right.

shell
abbr: *sh*

A new shell is created. When it terminates, editing resumes.

source <file>
abbr: *so*

Reads and executes commands from the specified file. Source commands can be nested.

substitute/<pattern>/<repl>/<options> <count> <flags>
abbr: *s*
Default: current line

On each specified line, the first instance of <pattern> is replaced by the replacement pattern <repl>. If the global indicator option character *g* appears, all instances are substituted. If the **confirm** indication character *c* appears, you are asked to confirm each substitution beforehand. The line to be substituted is typed (with the string to be substituted marked with ' ' characters). To accept the substitution, type *y*. Any other input causes no change to take place. After a *substitute* the current line is the last line substituted.

Lines can be split by substituting new-line characters into the line. The newline in <repl> must be escaped by preceding it with a backslash (\). Other metacharacters available in <pattern> and <repl> are described below.

stop

Suspends the editor, returning control to the top level shell. If autowrite is set and there are unsaved changes, a write is done first unless the form *stop!* is used. This command is only available when supported by the terminal driver and operating system.

(...) *substitute* <options> <count> <flags>
abbr: *a*

If *pat* and *repl* are omitted, then the 1st substitution is repeated. This is a synonym for the *&* command.

t <address> <flags>
Default: current line

The *t* command is a synonym for *cpy*.

ta <tag>

The focus of editing switches to the location of <tag>, changing to a different line in the current file where <tag> is defined or, if necessary, to another file.

The <tags> file is normally created by a program such as *ctags*, and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form that can be used by the editor to find the tag; this field is usually a contextual scan using /<pattern>/ to maintain immunity from minor changes in the file. Such scans are always performed as if *nomagic* were set.

The <tag> names in the *tags* file must be sorted alphabetically.

unabbreviate <word>

abbr: *una*

Delete <word> from the list of abbreviations.

undo

abbr: *u*

Reverses the changes made in the buffer by the last buffer editing command. Note that *global* commands are considered a single command for the purpose of *undo* (as are *open* and *visual*). Also, the commands *write* and *edit* (which interact with the file system) cannot be undone. *Undo* is its own inverse.

Undo always marks the previous value of the current line (.) as (“). After an *undo* the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect such as *global* and *visual*, the current line regains its pre-command value after an *undo*.

unmap <lhs>

The macro expansion associated by *map* for <lhs> is removed.

v/<pattern>/<commands>

Default: entire text

A synonym for the global command variant *g!*, running the specified <commands> on each line which does **not** match <pattern>.

version

abbr: *ve*

Prints the current version number of the editor as well as the date the editor was last changed.

visual <type> <count> <flags>

abbr: *vi*

Default: current line

Enters *visual* mode at the specified line. Type is optional and may be “-”, “ ”, or “.” as in the *z* command to specify the placement of the specified line on the screen. By default, if type is omitted, the specified line is placed as the first on the screen. A count specifies an initial window size; the default is the value of the option window. See the document *The Vi Editor* for more details. To exit this mode, type *q*.

visual <file>
or
visual + *n* <file>
write <file>
abbr *w*
Default: entire file

From visual mode, this command is the same as edit.

Writes changes made back to file, printing the number of lines and characters written. Normally file is omitted and the text goes back where it came from. If a file is specified, then text will be written to that file. If the file does not exist, it is created. The current file name is changed only if there is no current file name; the current line is never changed. If an error occurs while writing the current and edited file, the editor considers that there has been "no write since last change" even if the buffer had not previously been modified.

write>>><file>
abbr: *w*>>>
Default: entire file

Writes the buffer contents at the end of an existing file.

w! <name>

Overrides the checking of the normal write command, and will write to any file which the system permits.

w !<command>
Default: entire file

Writes the specified lines into command. Note the difference between *w!* which overrides checks and *w !* which writes to a command.

wq <name>

Like a write and then a quit command.

wq! <name>

The variant overrides checking on the sensibility of the write command, as *w!* does.

xit <name>

If any changes have been made and not written, writes the buffer out. Then, in any case, quits.

yank <buffer> <count>
abbr *ya*
Default: current line only

Places the specified lines in the names buffer, for later retrieval via *put*. If no buffer name is specified, the lines go to a more volatile place; see the *put* command description.

z <count>
Default: next line

Print the next count lines, default window.

z <type> <count>
Default: current line

Prints a window of text with the specified line at the top. If type is '-' the line is placed at the bottom; a '.' causes line to be placed in the center. A count gives the number of lines to be displayed rather than double the number specified by the scroll option. On a CRT the screen is cleared before display begins unless a count which is less than the screen size is given. The current line is left at the last line printed.

!`<command>`

The remainder of the line after the ‘!’ character is sent to a shell to be executed. Within the text of command the characters ‘%’ and ‘#’ are expanded as in filenames and the character ‘!’ is replaced with the text of the previous command. Thus, in particular, ‘!!’ repeats the last such shell escape. If any such expansion is performed, the expanded line will be echoed. The current line is unchanged by this command.

If there has been “no write” of the buffer contents since the last change to the editing buffer, then a diagnostic will be printed before the command is executed as a warning. A single ‘!’ is printed when the command completes.

`(<address>,<address>)!<command>`

Takes the specified address range and supplies it as standard input to the command; the resulting output then replaces the input lines.

`($) =`

Prints the line number of the addressed line. The current line is changed.

`(..)> <count> <flags>`

`(..)< <count> <flags>`

Perform intelligent shifting on the specified lines; `<` shifts left and `>` shifts right. The quantity of shift is determined by the *shiftwidth* option and the repetition of the specification character. Only white space (blanks and tabs) is shifted; no non-white characters are discarded in a left-shift. The current line becomes the last line which changed due to the shifting.

`~D`

An end-of-file from a terminal input scrolls through the file. The scroll option specifies the size of the scroll, normally a half screen of text.

`(+ 1,.. + 1)`

`(+ 1,.. + 1)|`

An address alone causes the addressed lines to be printed. A blank line prints the next line in the file.

`(..) & <options> <count> <flags>`

Repeats the previous *substitute* command.

`(..) ~ <options> <count> <flags>`

Replaces the previous regular expression with the previous replacement pattern from a substitution.

Regular Expressions and Substitute Replacement Patterns

Regular Expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be matched by the regular expression. *Ex* remembers two previous regular expressions; the previous regular expression used in a substitute command the the previous regular expression used elsewhere (referred to as a previous scanning regular expression). The previous regular expression can always be referred to by a null *re*, e.g. '/' or '??'.

Magic and Nomagic

The regular expressions allowed by *ex* are constructed in one of two ways depending on the setting of the *magic* option. The *ex* and *vi* default setting of *magic* gives quick access to a powerful set of regular expression metacharacters. The disadvantage of *magic* is that the user must remember that these metacharacters are "magic" and precede them with the character '\ ' to use them as "ordinary" characters. With *nomagic*, the default for *edit*, regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the (now) ordinary character with a '\ '. Note that '\ ' is thus always a metacharacter.

The remainder of this discussion of regular expressions assumes that the setting of this option is *magic*.

Summary of Basic Regular Expressions

The following basic constructs are used to construct magic-mode regular expressions.

<character>	An ordinary character matches itself. The characters (^) at the beginning of a line, (\$) at the end of line, (") as any character other than the first, (.), (\), ([) and (-) are not ordinary characters and must be escaped (preceded) by '\ ' to be treated as such.
^	At the beginning of a pattern forces the match to succeed only at the beginning of a line.
\$	At the end of a regular expression forces the match to succeed only at the end of the line.
.(period)	Matches a single character except the new-line character.
\<	Forces the match to occur only at the beginning of a "variable" or "word"; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.

- `\>` Similar to “`\<`” but matching the end of a “variable” or “word”, i.e. either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.
- `<string>` Matches any (single) character in the class defined by `<string>`. Most characters in string define themselves. A pair of characters separated by “-” in string defines the set of characters collating between the specified lower and upper bounds; thus `[a-z]` as a regular expression matches any (single) lower-case letter. If the first character of string is an `^` then the construct matches those characters which it otherwise would not; thus `[^a-z]` matches anything but a lower-case letter (and of course a newline). To place any of the characters `^`, `[`, or `-` in string you must escape them with a preceding `\`.

Combining Regular Expression Primitives

The concatenation of two regular expressions matches the leftmost and then longest string which can be divided with the first piece matching the first regular expression and the second piece matching the second. Any of the (single-character-matching) regular expressions mentioned above may be followed by the character `*` to form a regular expression which matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows.

The character `~` may be used in a regular expression, and matches the text which defined the replacement part of the last substitute command. A regular expression may be enclosed between the sequences `\(` and `\)` with side effects in the *substitute* replacement patterns.

Substitute Replacement Patterns

The basic metacharacters for the replacement pattern are `&` and `~`; these are given as `\&` and `\~` when *nomagic* is set. Each instance of `&` is replaced by the characters which the regular expression matched. The metacharacter `~` stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by the escaping character `\`. The sequence `\n` is replaced by the text matched by the *n*th regular subexpression enclosed between `\(` and `\)`. The sequences `\u` and `\l` cause the immediately following character in the replacement to be converted to upper- or lower-case respectively if this character is a letter. The sequences `\U` and `\L` turn such conversion on, either until `\E` or `\e` is encountered, or until the end of the replacement pattern.

Options Descriptions

autoindent

abbr: *ai*

Default: *noai*

Can be used to ease the preparation of structured program text. At the beginning of each *append*, *change*, or *insert* command or when a new line is opened or created by an *append*, *change*, *insert*, or *substitute* operation within *open* or *visual* mode, *ex* looks at the line being appended after, the first line changed or the line inserted before and calculates the amount of white space at the start of the line. It then aligns the cursor at the level of indentation so determined.

If the user then types lines of text in, they will continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line will start aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop one can hit $\wedge D$ (CNTRL-D). The tab stops going backwards are defined at multiples of the *shiftwidth* option. You cannot backspace over the indent, except by sending an end-of-file with a $\wedge D$.

Specially-processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the autoindent is discarded). Also specially processed in this mode are lines beginning with an \wedge and immediately followed by a $\wedge D$. This causes the input to be repositioned at the beginning of the line, but retaining the previous indent for the next line. Similarly, a 'O' followed by a $\wedge D$ repositions at the beginning but without retaining the previous indent.

Autoindent doesn't happen in *global* commands or when the input is not a terminal.

autoprint

abbr: *ap*

Default: *ap*

Causes the current line to be printed after each *delete*, *copy*, *join*, *move*, *substitute*, *l*, *undo*, or *shift* command. This has the same effect as supplying a trailing 'p' to each such command. *Autoprint* is suppressed in globals, and only applies to the last of many commands on a line.

autowrite

abbr: *aw*

Default: *noaw*

Causes the contents of the buffer to be written to the current file if you have modified it and give a *next*, *rewind*, *stop*, *tag*, or *!* command, or a $\wedge \uparrow$ (switch files) or $\wedge |$ (tag goto) command in *visual*. Note that the *edit* and *ex* commands do not autowrite. In each case, there is a equivalent way of switching when *autowrite* is set to avoid the *autowrite* (*edit* for *next*, *rewind!* for *.l rewind*, *stop!*, *tag!* for *tag*, *shell* for *!*, and *:e#* and a *:ta!* command from within *visual*).

beautify
abbr: *bf*
Default: *nobeautify*

directory
abbr: *dir*
Default: *dir = /tmp*

edcompatible
(no abbr)
Default: *noedcompatible*

errorbells
abbr: *eb*
Default: *noeb*

hardtabs
abbr: *ht*
Default: *ht = 8*

ignorecase
abbr: *ic*
Default: *noic*

lisp
(no abbr)
Default: *nolisp*

list
(no abbr)
Default: *nolist*

magic
(no abbr)
Default: *magic for ex and vi*

mesg
(no abbr)
Default: *mesg*

number
abbr: *nu*
Default: *nonumber*

Causes all control characters except tab, newline and form-feed to be discarded from the input. A complaint is registered the first time a backspace is discarded. *Beautify* does not apply to command input.

Specifies the directory in which *ex* places its buffer file. If this directory is not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

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

Error messages are preceded by a bell. If possible, the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

All upper case characters in the text are mapped to lower case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.

Autoindent indents appropriately for *lisp* code, and the *{}* *[* and *]* commands in *open* and *visual* are modified to have meaning for *lisp*.

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

If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only *|* and *\$* having special effects. In addition the metacharacters *-* and *&* of the replacement pattern are treated as normal characters. All the normal metacharacters may be made magic when *nomagic* is set by preceding them with a **.

Causes write permission to be turned off to the terminal while you are in *visual* mode, if *nomesg* is set.

Causes all output lines to be printed with their line numbers. In addition, each input line will be prompted for by supplying the line number it will have.

open
(no abbr)
Default: open

optimize
abbr: *opt*
Default: optimize

paragraph
abbr: *para*
Default: para = IPLPPPQPP LIbp

prompt
(no abbr)
Default: prompt

redraw
(no abbr)
Default: noredraw

remap
(no abbr)
Default: remap

report
(no abbr)
Default: report = 5 (2 for *edit*)

scroll
(no abbr)
Default: scroll = 1/2 window

sections
(no abbr)
Default: sections = SHNHH HU


shell
abbr: *sh*
Default: sh = /bin/sh

shiftwidth
abbr: *sw*
Default: sw = 8

If *noopen*, the commands *open* and *visual* are not permitted. This is set for *edit* to prevent confusion resulting from accidental entry to *open* or *visual* mode.


Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

Specifies the paragraphs for the {and} operations in *open* and *visual*. The pairs of characters in the option's value are the names of the macros which start paragraphs.

Command mode input is prompted for with a ':'.



The editor simulates (using great amounts of output) an intelligent terminal on a dumb terminal (e.g. during insertions in *visual* the characters to the right of the cursor position are refreshed as each input character is typed). Useful only at very high speed.

If on, macros are repeatedly tried until they are unchanged. For example, if o is mapped to O, and O is mapped to I, then if *remap* is set, o will map to I, but if *noremap* is set, it will map to O.

Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as *global*, *open*, *undo*, and *visual* which have potentially more far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a *global* command on the individual commands performed.


Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command-mode command (double the value of *scroll*).

Specifies the section macros for the [[and]] operations in *open* and *visual*. The pairs of characters in the option's value are the names of the macros which start paragraphs.

Gives the path name of the shell forked for the shell escape command '!', and by the *shell* command. The default is taken from SHELL in the environment, if present.


Gives the width a software tab stop, used in reverse tabbing with ^D when using *autoindent* to append text, and by the shift commands.

showmatch
abbr: *sm*
Default: nosm

slowopen
abbr: *slow*
Terminal dependent

tabstop
abbr: *ts*
Default: *ts* = 8

taglength
abbr: *tl*
Default: *tl* = 0

tags
(no abbr)
Default: *tags* = *tags/usr/lib/tags*

term
(no abbr)
From environment *TERM*

terse
(no abbr)
Default: not invoked

warn
(no abbr)
Default: *warn*

window
(no abbr)
Default: *window* = speed dependent

w300, *w1200*, *w9600*
(no abbr)
Default: not invoked

wrapscan
abbr: *ws*
Default: *ws*

wrapmargin
abbr: *wm*
Default: *wm* = 0

writeln
abbr: *wa*
Default: *nowa*

In *open* and *visual* mode, when a *)* or *}* is typed, move the cursor to the matching *(* or *{* for one second if this matching character is on the screen. Extremely useful with *lisp*.

Affects the display algorithm used in *visual* mode, holding off display updating during input or new text to improve throughput when the terminal in use is both slow and unintelligent. See *The Vi Editor* for more details.

The editor expands tabs in the output file to be on tabstop boundaries for the purposes of display.

Tags are not significant beyond this many characters. A value of zero (the default) means that all the characters are significant.

A path of files to be used as tag files for the *tag* command. A requested tag is searched for in the specified files, sequentially. By default (even in version 2) files called *tags* are searched for in the current directory and in */usr/lib* (a master file for the entire system).

The terminal type of the output device.

Shorter error diagnostics are produced for the experienced user.

Warn if there has been 'no write since last change' before a *!* command escape.

The number of lines in a text window in the *visual* command. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

These are not true options but set *window* only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.

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

Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. See *The Vi Editor* in this volume of *HP-UX Concepts and Tutorials* for more details.

Inhibit the checks normally made before *write* commands, allowing a write to any file which the system protection mechanism will allow.

Limitations

Editor limits that the user is likely to encounter are as follows: 1024 characters per line, 256 characters per global command list, 128 characters per file name, 128 characters in the previous inserted and deleted text in *open* or *visual*, 100 characters in a shell escape command, 63 characters in a string valued option, and 30 characters in a tag name, and a limit of 250000 lines in the file is silently enforced.

The *visual* implementation limits the number of macros defined with *map* to 32, and the total number of characters in macros to less than 512.

Ex Changes - Version 3.1 to 3.5

This update describes the new features and changes which have been made in converting from version 3.1 to 3.5 of *ex*. Each change is marked with the first version where it appeared.

Update to Ex Reference Manual

Command Line Options

A new command called *view* has been created. *View* is just like *vi* but it sets *readonly*. The encryption code from the v7 editor is now part of *ex*. You can invoke *ex* with the *-x* option and it will ask for a key, as *ed*. The *ed x* command (to enter encryption mode from within the editor) is not available. The feature may not be available in all instances of *ex* due to memory limitations.

Commands

Provisions to handle the new process stopping features of the Berkeley TTY driver have been added. A new command, *stop*, takes you out of the editor cleanly and efficiently, returning you to the shell. Resuming the editor puts you back in *command* or *visual* mode, as appropriate. If *autowrite* is set and there are outstanding changes, a write is done first unless you say "stop!".

A

```
:vi <file>
```

command from *visual* mode is now treated the same as a

```
:edit<file> or :ex<file>
```

command. The meaning of the *vi* command from *ex* command mode is not affected.

A new command mode command *xit* (abbreviated *x*) has been added. This is the same as *wq* but will not bother to write if there have been no changes to the file.

Options

A read only mode now lets you guarantee you won't clobber your file by accident. You can set the on/off option *readonly* (*ro*), and writes fail unless you use an *!* after the *write*. Commands such as *x*, *ZZ*, the *autowrite* option, and in general anything that writes is affected. This option is turned on if you invoke *ex* with the *-R* flag.

The *wrapmargin* option is now usable. The way it works has been completely revamped. Now if you go past the margin (even in the middle of a word) the entire word is erased and rewritten on the next line. This changes the semantics of the number given to *wrapmargin*. 0 still means off. Any other number is still a distance from the right edge of the screen, but this location is now the right edge of the area where wraps can take place, instead of the left edge. *Wrapmargin* now behaves much like *fill/justify* mode in *nroff*.

The options *w300*, *w1200*, or *w9600* can be set. They are synonyms for *window*, but only apply at 300, 1200, or 9600 baud, respectively. Thus you can specify that you want a 12 line window at 300 baud and a 23 line window at 1200 baud in your EXINIT with

```
:set w300=12w1200=23
```

The new option *timeout* (default on) causes macros to time out after one second. Turn it off and they will wait forever. This is useful if you want multi-character macros, but if your terminal sends escape sequences for arrow keys, it will be necessary to hit escape twice to get a beep. The new option *remap* (default on) causes the editor to attempt to map the result of a macro mapping again until the mapping fails. This makes it possible, say, to map q to # and #1 to something else and get ql mapped to something else. Turning it off makes it possible to map ^L to l and map ^R to ^L without having ^R map to l.

The new (string) valued option *tags* allows you to specify a list of tag files, similar to the "path" variable of *csh*. The files are separated by spaces (which are entered preceded by a backslash) and are searched left to right. The default value is "tags/user/lib/tags", which has the same effect as before. It is recommended that "tags" always be the first entry. On Ernie CoVax, /usr/lib/tags contains entries for the system defined library procedures from section 3 of the manual.

Environment Enquiries

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

Vi Tutorial Update

Deleted features

The "q" command from *visual* no longer works at all. You must use "Q" to get to *ex* command mode. The "q" command was deleted because of user complaints about hitting it by accident too often.

The provisions for changing the window size with a numeric prefix argument to certain *visual* commands have been deleted. The correct way to change the window size is to use the *z* command, for example *z5<cr>*, to change the window to 5 lines.

The option "mapinput" is dead. It has been replaced by a much more powerful mechanism: "map!".

Change in Default Option Settings

The default window sizes have been changed. At 300 baud the window is now 8 lines (it was 1/2 the screen size). At 1200 baud the window is now 16 lines (it was 2/3 the screen size, which was usually also 16 for a typical 24-line CRT). At 9600 baud the window is still the full screen size. Any baud rate less than 1200 behaves like 300, any over 1200 like 9600. This change makes *vi* more usable on a large screen at slow speeds.

Vi Commands

The command “ZZ” from *vi* is the same as “:x<cr>”. This is the recommended way to leave the editor. Z must be typed twice to avoid hitting it accidentally.

The command “Z” is the same as “:stop<cr>”. Note that if you have an arrow key that sends ^Z the stop function will take priority over the arrow function. If you have your “susp” character set to something besides ^Z, that key will be honored as well.

It is now possible from *visual* to string several search expressions together separated by semicolons the same as command mode. For example, you can say

```
/foo;/bar
```

from *visual* and it will move to the first “bar” after the next “foo”. This also works within one line.

^R is now the same as ^L on terminals where the right arrow key sends ^L (this includes the Televideo 912/920 and the ADM 31 terminals).

The *visual* page motion commands ^F and ^B now treat any preceding counts as number of pages to move, instead of changes to the window size. That is, 2^F moves forward 2 pages.

Macros

The “mapinput” mechanism of version 3.1 has been replaced by a more powerful mechanism. An “!” can follow the word “map” in the *map* command. *Map!*ed macros only apply during *input* mode, while *map*’ed macros only apply during *command* mode. Using “map” or “map!” by itself produces a listing of macros in the corresponding mode.

A word abbreviation mode is now available. You can define abbreviations with the *abbreviate* command:

```
:abbr foo find outer otter
```

which maps “foo” to “find outer otter”. Abbreviations can be turned off with the *unabbreviate* command. The syntax of these commands is identical to the *map* and *unmap* commands, except that the ! forms do not exist. Abbreviations are considered when in *visual* input mode only, and only affect whole words typed in, using the conservative definition. (Thus “foobar” will not be mapped as it would using “map!”) *Abbreviate* and *unabbreviate* can be abbreviated to “ab” and “una”, respectively.

Table of Contents

The Vi Editor

Preliminary Notes	1
Creating an Ordinary File	2
Invoking Vi	3
Moving around in the File	4
Cursor-Positioning Keys	6
Scrolling and Paging	6
Moving From Line to Line	7
Skipping Over Sentences, Paragraphs, and Sections	8
Searching for a Pattern	8
Moving Within a Line	11
Returning to Your Previous Position	12
Adding, Deleting, and Correcting Text	13
Inserting and Appending Text	14
Character Corrections	15
Line Corrections	16
Copying and Moving Text	18
Shifting Lines	19
Continuous Text Input	19
Undoing a Command	19
Special Vi Commands	20
Setting Vi Options	20
Defining Macros	24
Defining Abbreviations	25
Reading Data Into Your Current File	26
Writing Edited Text Onto a File	27
Editing Other Files	28
Editing the Next File in the Argument List	29
Filtering Buffer Text Through HP-UX Commands	30
Vi and Ex	31
The Shell Interface	31
Getting Into Vi	31
Getting Back to the Shell	32
Miscellaneous Topics	33
Vi Initialization	33
Recovering Lost Lines	34
Entering Control Characters in Your Text	34
Adjusting the Screen	35
Printing Your File Status	35
Appendix A: Character Functions	36
Appendix B: Example .exrc File	42





The Vi Editor

Vi is a display-oriented, interactive text editor. The contents of your file are displayed on your screen, so you can see the result of each *vi* command as soon as the command is executed. There is rarely any doubt about the current state of your file.

Preliminary Notes

Vi has two peculiar traits that might prove somewhat confusing to the beginning user. The first is that many of your commands do not print on your terminal when you type them in. Be assured that *vi* is still listening to you, however. If you watch the screen when you type in a command, *vi* usually gives some indication that your command has been received and interpreted. More specifically, the only commands that will print on your terminal are those that begin with /, :, ?, and !. If these characters are embedded in a long string of commands, only those characters after and including one of those above will be printed.

The second trait is that *vi* always uses the bottom line of the screen for command output, error messages, and echoed command lines. This is where you should look for information and command verification.

Creating an Ordinary File

The remainder of this article discusses the various commands and features of the *vi* editor. Because many *vi* commands do not print on the screen when they are executed, it is difficult to represent the results that appear on your screen before and after a command has executed. Thus, this article is designed to be read while you have access to a computer so you can try each command as it is discussed.

To be able to try each command, you need a file with some text in it. To create a file, type

```
$ vi filename
```

where *filename* is the name of the file you are creating. This file name is completely up to you. *Vi* responds by printing

```
"filename" [new file]
```

at the bottom of your screen, and prints a tilde (~) at the beginning of each line on the screen. The tilde is a special character that *vi* uses to mark the end of the text in a file that already exists, or, in the case of a new file, to show that there is currently no text in the file. The tildes are simply markers that are used for your convenience; they do not become part of the text in your file.

You are now ready to put text in your file. To do this, type **a** (for append). Even though the command does not print on your screen, *vi* is now waiting for your text. As you type in your text, note that everything you type appears on your screen, and that the tilde on each line disappears as you begin typing on that line.

It does not really matter what you type in for your text, but you need at least two paragraphs of material (paragraphs must be separated by at least one blank line). That amount of text ensures that most of the commands can be illustrated on your file. When you are done entering text, press [ESC], and exit the editor by typing **ZZ**. You should now have a shell prompt on your screen.

Invoking Vi

Material Covered:

<code>vi file ...</code>	command: invokes <i>vi</i> with one or more file arguments;
<code>[ESC], [ALT], ctrl-[</code>	commands: end text insertion or modification;
<code>[DEL], [RUB], ctrl-?</code>	commands: generate an interrupt.

You invoke *vi* the same way you invoke any shell command. *Vi* accepts several options and a list of file names, which are the names of the files you want to create or edit. For a list of the available options, refer to the HP-UX Reference manual. For example,

```
$ vi file1 file2 file3
```

invokes *vi* with `file1`, `file2`, and `file3` as arguments. `file1` is created or edited first. *Vi* remembers `file2` and `file3` so that you can create or edit them after you are finished with `file1`. Begin editing the file you created previously by typing

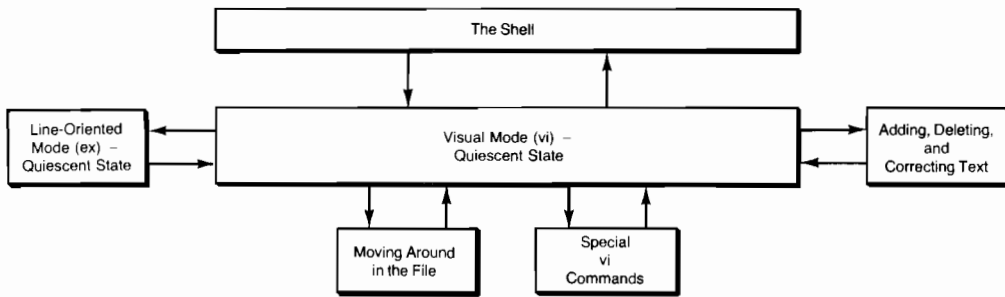
```
$ vi filename
```

where *filename* is the name of the file you created. Note that *vi* prints out either a screenful of text from *filename*, or the entire contents of *filename* followed by a tilde on each remaining empty line. *Vi* does the latter if *filename* does not contain enough text to fill the screen. Your cursor is positioned at the beginning of the first line of the file. *Vi* is now waiting for your commands.

Vi always copies the contents of the file you are editing into a special buffer. All additions, deletions, and corrections are performed on the copy in the buffer. This way, the original file remains unchanged until you are sure you want to change it. Then, when you are finished editing the file, you can tell *vi* to overwrite the previous contents of the file with the revised text in the buffer. Even if you are creating a file, the text you put in your file is actually put in the buffer. The text remains there until you tell *vi* to transfer it to the file you are creating.

Once you have invoked *vi*, it enters a do-nothing state in which it waits for a command. This is called a *quiescent state*. You can determine what state *vi* is in by pressing `[ESC]` or `[DEL]`. `[ESC]` is used to end text insertion and to cancel partially formed commands. If you press `[ESC]` and *vi* responds by ringing the bell, then *vi* is in a quiescent state. If *vi* does not ring the bell, then it is busy executing a command. `Ctrl-[` generates the same sequence as the `[ESC]` or `[ALT]` key on your keyboard. `[DEL]` generates an interrupt, which forces *vi* to stop whatever it is doing and return to a quiescent state.

Once *vi* is in a quiescent state, there are several things you can do. They are shown in the following diagram.



Moving Around in the File

Material Covered:

[↑], k	commands: move the cursor up one line in the same column:
[→], l , [SPACE]	commands: move the cursor one character to the right:
[↓], j , ctrl-J , ctrl-N	commands: move the cursor down one line in the same column:
[←], h , [BACKSPACE], ctrl-H	commands: move the cursor one character to the left:
ctrl-D	command: scroll down:
ctrl-U	command: scroll up;
ctrl-E	command: scroll up one line:
ctrl-Y	command: scroll down one line:
ctrl-F	command: move forward one page in the file:
ctrl-B	command: move backward one page in the file:
+ , [RETURN], ctrl-M	commands: move the cursor to the first printable character on the next line:
-	command: move cursor to the first printable character on the previous line:
nG	command: move cursor to first printable character on line number <i>n</i> : default <i>n</i> = last line of the file:
H	command: move cursor to the first printable character of the first line on the screen:
M	command: move cursor to the first printable character of the middle line on the screen:
L	command: move cursor to the first printable character of the last line on the screen:
m	command: mark a particular line with a label:
%	command: show matching left or right parenthesis or brace:
(command: move cursor to the beginning of the most previous sentence:
)	command: move cursor to the beginning of the next sentence:

{	command: move cursor to the beginning of the most previous paragraph;
}	command: move cursor to the beginning of the next paragraph;
[[command: move cursor to the beginning of the most previous section;
]]	command: move cursor to the beginning of the next section;
/	command: initiates a forward pattern search;
?	command: initiates a backward pattern search;
n	command: repeats the most previous pattern search;
N	command: repeats the most previous pattern search in the opposite direction;
^	metacharacter: used in pattern searches to match a pattern at the beginning of a line;
\$	metacharacter: used in pattern searches to match a pattern at the end of a line;
\	metacharacter: used in pattern searches to strip away the special meaning of a metacharacter;
.	metacharacter: used in pattern searches to match any single character;
\<	metacharacter: used in pattern searches to match a pattern at the beginning of a word;
\>	metacharacter: used in pattern searches to match a pattern at the end of a word;
[...]	metacharacters: used in pattern searches to match any one of the enclosed characters;
*	metacharacter: used in pattern searches to match zero or more instances of the preceding character;
w	command: move cursor forward to the beginning of the next word, or to the next punctuation mark, whichever comes first;
W	command: move cursor forward to the beginning of the next word, ignoring punctuation;
b	command: move cursor backwards to the beginning of the previous word, or to the most previous punctuation mark, whichever comes first;
B	command: move cursor backwards to the beginning of the previous word, ignoring punctuation;
e	command: move cursor forward to the end of the next word, or to the next punctuation mark, whichever comes first;
E	command: move cursor forward to the end of the next word, ignoring punctuation;
fc	command: move cursor forward to the next instance of the specified character, c;
Fc	command: move cursor backwards to the next instance of the specified character, c;
tc	command: move cursor forward to the first character to the left of the next instance of the specified character, c;

Tc	command: move cursor backwards to the first character to the right of the next instance of the specified character, <i>c</i> ;
:	command: repeats the most previous f , F , t , or T command;
,	command: repeats the most previous f , F , t , or T command, in the opposite direction;
^ , 0 (zero)	commands: move cursor to the first printable character on the current line;
\$	command: move cursor to the end of the current line;
 	command: move cursor to specified column number in current line;
`` or ''	commands: returns cursor to its most previous position.

This section describes several commands that enable you to move around in your file. You should try each of these commands as they are discussed to familiarize yourself with them.

Cursor-Positioning Keys

If your terminal has cursor-positioning keys, these keys can be used in *vi* to position the cursor in the file you are editing. The **h**, **j**, **k**, and **l** commands perform the same functions as the cursor-positioning keys. The **h** command moves the cursor one space to the left ([BACKSPACE] and ctrl-H also moves the cursor one space to the left). The **j** command moves the cursor down one line in the same column (as do ctrl-J and ctrl-N), the **k** command moves the cursor up one line in the same column (as does ctrl-P), and the **l** command moves the cursor one space to the right ([SPACE] also moves the cursor one space to the right). These commands are summarized below:

$$\begin{aligned}
 [\uparrow] &= \mathbf{k} \\
 [\rightarrow] &= \mathbf{l} = [\text{SPACE}] \\
 [\downarrow] &= \mathbf{j} = \text{ctrl-J} = \text{ctrl-N} \\
 [\leftarrow] &= \mathbf{h} = [\text{BACKSPACE}] = \text{ctrl-H}
 \end{aligned}$$

Scrolling and Paging

The **ctrl-D** command scrolls down in the file, leaving several lines of continuity between the previous screenful of text and the new screenful of text (note that [CTRL] must be held down while the next key is pressed). The **ctrl-U** command scrolls up in the file, also leaving several lines of continuity on the screen. If either **ctrl-D** or **ctrl-U** is preceded by a number argument, then the number of lines scrolled is equal to that specified number, and remains so until changed again.

If you want more control over the scrolling process, the **ctrl-E** command exposes one more line at the bottom of the screen, and the **ctrl-Y** command exposes one more line at the top. Preceding **ctrl-E** or **ctrl-Y** with a number causes the command to be executed that many times.

There are two paging commands, **ctrl-F** and **ctrl-B**, which move forward and backward one page in the file, respectively. Both commands leave a few lines of continuity between screenfuls of text. Giving a number argument to either of these paging commands executes the command that many times.

Note that paging moves you more abruptly than scrolling does, and leaves you fewer lines of continuity between screenfuls of text.

Moving From Line to Line

The `+` and `-` commands move the cursor to the first printable character on the next line or the previous line, respectively. `[RETURN]` or `ctrl-M` have the same effect as `+`. A preceding number argument executes these commands that many times.

The `G` command, when preceded by a line number, positions the cursor at the beginning of that line in the file. For example, `3G` positions the cursor at the beginning of the third line. If you do not specify a number, the cursor is positioned at the beginning of the last line of the file.

The `H` command positions the cursor at the beginning of the first line on the screen. If you precede `H` with a number, as in `4H`, the cursor is positioned at the beginning of the fourth line on the screen.

The `M` command positions the cursor at the beginning of the middle line on the screen. The `M` command ignores any line number argument.

The `L` command positions the cursor at the beginning of the last line on the screen. You can precede the `L` command by a number, as in `4L`, which positions the cursor at the beginning of the fourth line above the bottom of the screen.

Note that the `H`, `M`, and `L` commands reference the first, middle, and last lines of the current screenful of text. They do not reference the first, middle, and last lines of the entire file.

The `m` command enables you to mark specific lines with a label so that you can return to them. The label must be a single, lower-case letter in the range "a" through "z". To mark a line, first move the cursor to the particular line (using any of the commands described in *Moving Around in the File*), and type `m?`, where `?` is the label you have selected. For example, `+++me` moves the cursor ahead three lines and marks that line with the label "e".

To reference a line you have marked, precede your label with a grave accent (```). For example, ``e` moves the cursor to the line you marked with the label "e". Note also that the cursor is placed in exactly the same spot within the line that it was when you marked the line. If you are not particularly interested in a specific position within a marked line, use an apostrophe (`'`) instead of a grave accent. Thus, `'e` moves the cursor to the beginning of the line marked by the label "e", regardless of where the cursor was in the line when you marked it. Try marking a few lines, using both the apostrophe and grave accent, until you are familiar with their differences.

Marks are defined until you begin editing another file, or until you leave the editor. Marks cannot be erased.

The `%` command shows you the matching left or right parenthesis or brace for the parenthesis or brace currently marked by the cursor.

Skipping Over Sentences, Paragraphs, and Sections

The (and) (left and right parentheses) commands move the cursor to the beginning of the previous and next sentences, respectively. A sentence is defined to end at a period, an exclamation point, or a question mark, followed either by two spaces or the end of a line. Any number of closing parentheses, brackets, double quotes, or single quotes may follow the period, exclamation point, or question mark, as long as they occur before the two spaces or the end of the line. The (and) commands can be preceded by a number to move the cursor over several sentences at once.

The { and } (left and right braces) commands move the cursor to the beginning of the previous and next paragraphs, respectively. A paragraph is defined as a block of text beginning and ending with a blank line, or a block of text delimited by macro invocations. The default list of macros (from the *-ms* and *-mm* macros packages) includes **.JP**, **.LP**, **.PP**, **.QP**, **.P**, **.LI**, and **.bp**. These macros are used so that files containing *nroff/troff* text can be easily edited with *vi*. You may add your own macro names to those already recognized by appropriately setting the **paragraphs** option (see *Setting Vi Options* later in this article). The { and } commands can be preceded by a number to move the cursor over several paragraphs at once.

The [[and]] (double left and right brackets) commands move the cursor to the beginning of the previous and next sections, respectively. A section is defined as beginning and ending with a line containing a ctrl-L (formfeed character) in the first column, or as a block of text delimited by macro invocations. The default list of macros defining a section is **.NH**, **.SH**, **.H**, and **.HU**. You may add your own macro names to those already understood by appropriately setting the **sections** option (see *Setting Vi Options* later in this article). If [[or]] is preceded by a number argument, it is interpreted to be the new window size (number of lines per screenful of text).

Searching for a Pattern

You can tell *vi* to search for a particular pattern (string of characters) in your file. To do this, type a slash (/), followed by the pattern you want to search for, followed by [RETURN]. Note that the entire command is printed at the bottom of your screen. If *vi* finds the pattern, *vi* positions the cursor at the beginning of the pattern. If the pattern cannot be found, *vi* prints an error message and returns the cursor to its location prior to the search.

The slash initiates a forward search, with wraparound, starting from the current position of the cursor. Replacing the slash with a question mark (?) initiates a backward search, with wraparound, starting from the current position of the cursor. If a number argument is specified before / or ?, it is interpreted to be the new window size (number of lines per screenful of text).

If you want your pattern to match only at the beginning of a line, begin your pattern with a caret (^). If you want your pattern to match only at the end of a line, end your pattern with a dollar sign (\$).

Here are some examples:

```
.test[RETURN]
```

This is a forward search for the string "test". Note that this pattern matches "re-test", "testing", "detestable", or "test". To find only the word "test" standing alone (but not at the end of a sentence, or just before a comma), type

```
test [RETURN]
```

The spaces require that "test" not be part of another word.

```
?^Today[RETURN]
```

This is a backward search for the string "Today" appearing only at the beginning of a line.

```
regret$[RETURN]
```

This is a forward search for the string "regret" appearing only at the end of a line.

The **n** command enables you to repeat the most recently executed search. Each time **n** is typed, the previous search is re-executed. The **N** command also repeats the most recently executed search, but in the opposite direction. These commands are handy for finding a particular occurrence of a pattern without having to re-type the search each time.

There are times when you want to position the cursor at the beginning of the line containing the pattern. This can be done by typing your search command in a slightly different way. For example,

```
key/+0[RETURN]
```

searches forward and positions the cursor at the beginning of the line containing the string "key". You can also position the cursor at the beginning of a line relative to the line containing the pattern. For example,

```
FIFO/-3[RETURN]
```

searches forward and positions the cursor at the beginning of the third line before the line containing the string "FIFO". Also,

```
?CRT?+2[RETURN]
```

searches backward and positions the cursor at the beginning of the second line after the line containing the string "CRT".

There are two options, **magic** and **nomagic**, which affect the way you can specify patterns (see the section entitled *Setting Vi Options*). If the **nomagic** option is set, then only the characters `^` and `$` have special meaning in patterns. If you want to include either of these characters in the actual pattern you search for, they must be preceded by a backslash (`\`). The backslash *quotes* the character immediately following it, and strips away any special meaning that character might have. For example,

```
^\L[RETURN]
```

searches for the string `^L`. The backslash was necessary to keep the caret from being interpreted to mean "match this pattern at the beginning of a line".

If the **magic** option is set, then you have several other special characters that you can use in patterns, including `^` and `$`. The `.` (dot) matches any character, as in

```
chap.[RETURN]
```

which matches any five-character string that begins with "chap". The character combinations `\<` and `\>` match the beginning of a word and the end of a word, respectively. For example,

```
?\<how[RETURN]
```

matches any word beginning with "how", including "how" itself. Also,

```
ed\>[RETURN]
```

matches any word ending with "ed", including "ed" itself.

Brackets are also special, and match any one of the characters enclosed in them. For example,

```
file[123][RETURN]
```

matches "file1", "file2", and "file3". If the characters inside the brackets are preceded by a `^`, then the brackets match any single character *not* enclosed in them, as in

```
chap[^1234][RETURN]
```

which matches any five-character string beginning with "chap", except "chap1", "chap2", "chap3", and "chap4". If you want to specify large spans of letters or numbers, as in a through z, or 0 through 9, they can be abbreviated inside the brackets, as in `[a-z]` or `[0-9]`.

The asterisk (`*`) matches zero or more instances of the character immediately preceding it. For example,

```
b*[RETURN]
```

matches zero or more b's. Note that this is a useless search, since zero b's can be found much quicker than one or more b's. To find one or more b's, you must type

```
/bb*[RETURN]
```

Also,

```
/[123][123]*[a-z][RETURN]
```

matches a one, two, or three, followed by any number of one's, two's, and three's, followed by a single lower-case letter. Experiment with the asterisk until you understand the implications of matching zero or more occurrences of a pattern.

If the **magic** option is set, then the characters `^`, `$`, `.`, `\<`, `\>`, `[`, `]`, and `*` have special meaning and must be quoted with a backslash if you want them to be literally matched in a pattern (note that the characters `\<` and `\>` must each be preceded by a backslash, as in `\\<` and `\\>`). If the **nomagic** option is set, then only `^` and `$` require a backslash to be literally matched. Note that, to match a backslash literally, it also must be preceded with a backslash.

The characters `^`, `$`, `.`, `\<`, `\>`, `[`, `]`, `*`, and `\` are commonly called *metacharacters* whenever their special meanings are utilized. This helps to distinguish between their normal, literal use, and their use as special characters.

Moving Within a Line

The **w** and **W** commands advance the cursor to the beginning of the next word in the sentence, wrapping around to the next line if necessary. The difference between the two commands is that the **w** command also stops at each punctuation mark it encounters; the **W** command does not stop at punctuation.

The **b** and **B** commands move the cursor backwards to the beginning of the previous word, wrapping around to the previous line if necessary. The **b** command stops at punctuation, while the **B** command does not.

The **e** and **E** commands advance the cursor to the end of the next word in the sentence, wrapping around to the next line if necessary. The **e** command stops at punctuation, while the **E** command does not.

Note that the **w**, **W**, **b**, **B**, **e**, and **E** commands all wrap around to lines other than the current line. These commands can be preceded by a number to move the cursor over several words at once.

The **f** and **F** commands move the cursor forward or backward, respectively, to the next occurrence of the specified character. The cursor is placed on the specified character. For example, **fc** moves the cursor forward to the first occurrence of the character "c", and **F:** moves the cursor backwards to the first occurrence of a colon. The **f** and **F** commands can be preceded by a number, as in **3fr**, which moves the cursor forward to the third occurrence of the character "r". Both **f** and **F** work only on the current line, and do not wrap around to other lines.

The **t** and **T** commands are identical to the **f** and **F** commands, except that the cursor is placed one character to the left or right of the specified character, respectively. For example, **2Tm** moves the cursor backwards to the second occurrence of the character "m", and places the cursor one character to the right. **3t.** moves the cursor forward to the third occurrence of a period, and places the cursor one character to the left.

The **:** command repeats the most previously executed **f**, **F**, **t**, or **T** command. Thus, **fi;;** is identical to **4fi**, and **Tj;** is identical to **2Tj**. The **,** command also repeats the most previously executed **f**, **F**, **t**, or **T** command, but in the opposite direction. Thus, if you execute **Tk**, a subsequent **,** searches forward in the current line for the letter k.

The **^** (caret) command moves the cursor to the first printable character on the current line. The **0** (zero) command is a synonym for **^**. Any number argument is ignored.

The **\$** command moves the cursor to the end of the current line. If a number argument *n* is specified, **\$** moves the cursor to the *n*th end of line it finds. Thus, **\$** can wrap around to other lines, but only if preceded by a number argument (note that several explicitly typed **\$**'s will not do this).

The **|** (vertical bar) command moves the cursor to the character in the column specified by a preceding number argument. If no number is given, **|** is a synonym for **^** and **0**, in that it moves the cursor to the first printable character in the line.

Note that the **f**, **F**, **t**, **T**, **^**, **0**, and **|** commands work only on the current line. If you want to use these commands on a line other than the current line, you must first move the cursor to the line of interest.

Returning to Your Previous Position

The **``** (two grave accents) command and the **''** (two apostrophes) command both return you to your previous position. These commands can be used after you have executed a search command or one of the commands listed under *Moving Around in the File*, and you want to get back to where you were. **V** remembers only your last previous position.

Adding, Deleting, and Correcting Text

Material Covered:

i	command: insert text before cursor;
I	command: insert text at the beginning of a line (same as ^i);
a	command: append text after cursor;
A	command: append text at the end of a line (same as \$a);
o	command: create new line below line containing cursor;
O	command: create new line above line containing cursor;
x	command: delete character marked by cursor;
X	command: delete character immediately before character marked by cursor;
r	command: replace character marked by cursor with another character;
s	command: replace one or more characters with one or more characters;
d	command: delete; can be combined with several other commands specifying what is to be deleted;
D	command: delete from current location through end of line (same as d\$);
c	command: change; can be combined with several other commands specifying what is to be changed;
C	command: change from current location through end of line (same as c\$);
.	command: re-execute last operation which changed text in buffer;
y	command: copy specified amount of text into a specified buffer;
Y, yy	commands: copy the specified number of complete lines into a specified buffer;
"	operator: introduces buffer name in which text is saved by previous y or Y commands;
a-z	buffers: the buffer names in which text can be saved with y or Y commands; there is, in addition, an unnamed buffer;
p	command: puts saved text back into the file, after or below the cursor;
P	command: puts saved text back into the file, before or above the cursor;
<<	command: shifts the specified number of lines one shift-width to the left;
>>	command: shifts the specified number of lines one shift-width to the right;
<	command: shifts the specified lines one shift-width to the left; can be combined with other commands;
>	command: shifts the specified lines one shift-width to the right; can be combined with other commands;
J	command: joins the specified number of lines together;
u	command: reverses the last change made to the file;
U	command: restores the current line back to its state before editing began;

Inserting and Appending Text

The **i** and **a** commands are used for inserting and appending text, respectively. The **i** command places text to the left of the cursor, and the **a** command places text to the right of the cursor. Both commands are cancelled by [ESC].

You may insert or append many lines of text, or just a few characters, with the **i** and **a** commands. To type in more than one line of text, press [RETURN] at the place in your text where you want the new line to appear. When you are inserting or appending text, [RETURN] causes *vi* to create a new line, and to copy the remainder of the current line onto the new line.

If a number *n* is specified before the **i** or **a** command, then the text you add is duplicated *n*-1 times when [ESC] is pressed. This works only if there is room on the current line for the duplications. For example, if you type **5a** at some particular point in a line, and your appended text is "hi", then, when [ESC] is pressed, the text actually appended will be expanded to "hihihihi".

If you want to start adding text on a new line that does not currently exist, you can create a new line in your text with the **o** and **O** commands. The **o** command creates a new line after the line containing the cursor, and the **O** command creates a new line before the line containing the cursor. The **o** and **O** commands can create only one new line, but pressing [RETURN] while using the **o** and **O** commands causes *vi* to create an additional new line for you. The **o** and **O** commands are cancelled by [ESC], and ignore any preceding number argument. Thus, the only difference between the **i**, **a**, **o**, and **O** commands is that the **o** and **O** commands automatically create a new line on which text can be added, while the **i** and **a** commands do not. New lines can be created with all four commands simply by pressing [RETURN].

During an insert or append operation, if a **ctrl-@** is typed as the first character of the text to be inserted/appended, the **ctrl-@** is replaced by the most previous text that was inserted or appended. A maximum of 128 characters are saved from the previous text addition. If more than 128 characters were inserted or appended in the last text addition, the **ctrl-@** function is not available during the current text addition.

If you are in insert or append mode, the **autoindent** option is set, and you are at the beginning of a line, **ctrl-T** causes **shiftwidth** white space to be inserted at that point. White space inserted in this manner can be back-tabbed over with **ctrl-D** in insert or append mode. **Ctrl-D** is necessary because **shiftwidth** white space cannot be backspaced over.

The **ctrl-W** sequence enables you to back up over words (similar to **b** in command mode) while in insert or append mode. All words backed over are deleted from the text addition, even though the characters still appear on your screen.

The keys you use at the shell level to erase characters or entire lines can also be used in *vi*. When you are inserting or appending text, single characters can be erased with [BACKSPACE], and entire lines can be erased with **ctrl-U**. (Note that [BACKSPACE] and **ctrl-U** are the default keys assigned to erase single characters and entire lines. Your keys may have been re-defined. Check with your system administrator.) Note that you cannot erase characters which you did not insert or append, and that you cannot backspace into a previous line.

Experiment with the **i**, **a**, **o**, and **O** commands until you are familiar with what each command does. Be sure to note the effects of pressing [RETURN] with each of these commands.

Character Corrections

The **x** command deletes the character marked by the cursor. You can delete more than one character by preceding **x** with a number. **3x**, for example, deletes the next three characters, including the one marked by the cursor.

The **X** command deletes the character immediately before the one marked by the cursor. Preceding **X** with a number deletes that many characters before the current location of the cursor.

Both **x** and **X** work only on the current line; they cannot delete characters on any line other than the current line.

The **r** command replaces one character with another. For example, **rT** replaces the character marked by the cursor with the character "T". If a number *n* precedes the **r** command, then *n* characters are replaced by the single character you type next. For example, **4rt** replaces the next four characters with the letter t.

The **s** command replaces one or more characters with the specified string of characters. When not preceded by a number, the **s** command replaces a single character with the specified string. For example,

```
sTTY[ESC]
```

replaces the character marked by the cursor with the string "TTY". When preceded by a number, the **s** command replaces the specified number of characters, beginning with the character marked by the cursor, with the specified string of characters. For example,

```
4sinteresting[ESC]
```

replaces the next four characters with the string "interesting". Note that the **s** command prints a dollar sign at the end of the text to be replaced so you can see the extent of the change. The dollar sign is removed when you press [ESC].

The **d** command can be combined with several of the commands previously discussed to delete characters and words. For example, **dw** deletes the next word, and **db** deletes the previous word. **d[SPACE]** deletes the character marked by the cursor (this is equivalent to the **x** command). The **d** command can be preceded by a number to delete several words or characters, as in **3db**, which deletes the last three words. The **d** command can also be used with the **f**, **F**, **t**, and **T** commands. For example, **dtr** deletes everything from the current position of the cursor up to (but not including) the next "r" that appears in the current line. Experiment with these combinations until you are familiar with their effects.

The **c** command can also be combined with several other commands to change characters and words. The **c** command can be preceded by a number. Here are some examples:

```
c5yesterday[ESC]
```

This changes the next five words to the string "yesterday". Note that the "c" and the "5" could be interchanged with the same result.

```
4cbvariable name[ESC]
```

This changes the previous four words to the string "variable name".

```
c[SPACE]in a buffer[ESC]
```

This changes the character marked by the cursor to the string "in a buffer".

```
cfqHP-UX operating system[ESC]
```

This changes everything from the current position of the cursor up to (and including) the first occurrence of a "q" to the string "HP-UX operating system". The **c** command can be used similarly with the **F**, **t**, and **T** commands.

Note that the **c** command marks the end of the text to be changed with a dollar sign so you can see the extent of the change. The dollar sign is removed after you press [ESC].

Line Corrections

The **d** and **c** commands can also delete or change lines or groups of lines. The **d** command can be appended to itself to delete one complete line. For example, **dd** deletes the current line, and **5dd** deletes the current line and the next four lines.

The **d** command can be combined with several other commands. For example, **dL** deletes everything from the current position of the cursor through the last line on the screen. **d3L** deletes everything from the current position of the cursor through the third line from the bottom of the screen. The **d** command can also be used with a search, so that

```
d market$[RETURN]
```

deletes everything from the current position of the cursor up to the beginning of the string "market", which must occur at the end of a line. Try the **d** command with the (,), {, }, [[, and]] commands to delete one or more sentences, paragraphs, or sections.

Note that any of the commands discussed under *Moving Around in the File* can be combined with the **d** command to delete specific portions of text. Also note that, if you delete five or more lines, *vi* informs you of the number of lines deleted with a message at the bottom of your screen.

The **D** command is shorthand for **d\$**, causing all characters from the cursor to the end of the line to be deleted. Any preceding number argument is ignored.

The **c** command can also be appended to itself (thus creating the **cc** command) to change one complete line. **S** is a synonym for **cc**. For example,

```
ccEnter the value for variable A.[ESC]
```

changes the current line to the sentence "Enter the value for variable A."

```
4SPlace illustration here.[ESC]
```

This changes the current line and the three lines following it to the sentence "Place illustration here.". Note that the **S** command was used, and that the results are the same as if **cc** had been used.

```
cMPlace output on TTY4.[RETURN]Call exit routine.[ESC]
```

This changes everything from the current position of the cursor to the middle line on the screen to the two sentences "Place output on TTY4." and "Call exit routine.". Note that each sentence is on a separate line.

```
)c([RETURN]Insert new paragraph here.[RETURN])[ESC]
```

Here, the initial ")" moves the cursor to the beginning of the next paragraph, and then the entire previous paragraph is changed to a blank line, followed by the sentence "Insert new paragraph here.", followed by another blank line.

```
+c/while/-I[RETURN]continue:[ESC]
```

The initial "+" advances the cursor to the first printable character on the next line. Then, everything from the beginning of that line up to and including the line before the next "while" statement is changed to the single statement "continue:".

Like the **d** command, the **c** command can be combined with any of the commands discussed under *Moving Around in the File*, and *vi* informs you when five or more lines are being changed. Also, as in previous **c** examples, the end of the text to be changed is marked with a **\$**. Try some of the other combinations not covered above until you are familiar with how **c** works.

The **C** is equivalent to **c\$**, causing all the characters from the cursor to the end of the line to be changed to the text that follows. Any preceding number argument is ignored.

The **.** (dot) command repeats the last command which made a change in the text. Thus, **dw....** is the same as **6dw**, in that both commands delete the next six words. The dot command can be used to re-execute *any* command which modified the buffer text, but is limited to that command which was executed most recently.

Copying and Moving Text

The **y** command copies a specified portion of text into a buffer. There are 26 named buffers, named "a" through "z", and one unnamed buffer. If you do not specify a buffer name, the copied text is automatically placed in the unnamed buffer. For example, **yw** copies the next word into the unnamed buffer, and **y2B** copies the previous two words into the unnamed buffer.

When specifying a buffer name, the name must be preceded by a double quote ("). This tells *vi* that the character to follow is a buffer name. For example, **"ay2(** copies the previous two sentences into buffer "a" (the initial **)** ensures that complete sentences are copied). Also,

```
"ty:~two[RETURN]
```

copies everything from the current position of the cursor up to the line beginning with the string "two", and puts the text in buffer "t".

Note that the **y** command starts copying at the current position of the cursor. Thus, partial words or sentences may be copied if the cursor is in the middle of a word or sentence when you give the **y** command. Note also that, when copying forward in the file, the character marked by the cursor is included in the copied text. When copying backwards, however, the copied text begins with the character preceding the character marked by the cursor.

The **Y** command is used to copy complete lines of text, regardless of the position of the cursor within the line. For example, **3Y** copies three lines, including the current line, into the unnamed buffer. **"f6Y** copies six lines, including the current line, into buffer "f". A synonym for **Y** is **yy**.

The **p** and **P** commands put the copied text back into the file relative to the location marked by the cursor. The **p** command puts the text after or below the cursor, and the **P** command puts the text before or above the cursor. Exactly where the text is placed in relation to the cursor is determined by the amount of text being placed. If there is room on the current line for the text, then the text is placed after (**p**) or before (**P**) the cursor. If there is too much text to fit on one line, then *vi* creates one or more new lines below (**p**) or above (**P**) the cursor, and puts the text there. For example, **"rp** puts the text contained in buffer "r" into the file after or below the cursor. If no buffer name is specified, the text in the unnamed buffer is put back into the file.

Up to now, the copied text has been left in its original location and duplicated elsewhere in the file. If you do not want the text left in its original location, you can use the **d** command. For example, **5dd** deletes the next five lines, and saves them in the unnamed buffer (that's right - every deletion you perform is saved in the unnamed buffer until it is overwritten by the next deletion). **"wd2}** deletes the next two complete paragraphs (if the cursor is at the beginning of a paragraph) and saves them in buffer "w". The **p** or **P** command can then be used to put the deleted text elsewhere in the file.

You can copy text from one file into another. First, save the text in any of the named buffers. Once the text is saved, stop editing the current file and begin editing the file in which the text is to be inserted (the commands used to edit other files are described in the section entitled *Editing Other Files*). Now use the **p** or **P** command to put the saved text into the file. Do not use the unnamed buffer to transfer text from one file to another, because the contents of the unnamed buffer are lost when you change files.

Shifting Lines

The << and >> commands move the specified number of lines one shift-width to the left or right, respectively. One shift-width is equal to the number of columns specified by the **shiftwidth** option (see the section entitled *Setting Vi Options*). For example, **4>>** moves four lines one shift-width to the right. The << and >> commands are limited to numerical arguments only.

The < and > commands can be used with numbers and other commands to shift large groups of lines. For example, **>3L** moves every line from the current line to the third line from the bottom of the screen one shift-width to the right. Also,

```
</RAM[RETURN]
```

moves every line from the current line to the first line containing the string "RAM" one shift-width to the left. The < and > commands may be combined with any of the commands discussed under *Moving Around in the File*.

Continuous Text Input

When you are typing in large amounts of text, it is convenient to have your lines automatically broken and continued on the next line so that you do not have to press [RETURN]. The **wrapmargin** option enables you to do this (see the section entitled *Setting Vi Options*). For example, if the **wrapmargin** option is set equal to 10, *vi* breaks each line at least 10 columns from the right-hand edge of the screen.

If you want to join broken lines together, use the **J** command. For example, **3J** joins three lines together, beginning with the current line. *Vi* supplies white space at the place or places where the lines were joined, and moves the cursor to the first occurrence of the supplied white space.

Undoing a Command

The **u** command reverses the last change you made to your text. The **u** command is able to undo only the last change you have made. Note that a **u** command also undoes itself. If you have made several changes to a line, and you want to reverse *all* of the changes, use the **U** command. The **U** command restores the current line back to the state it was in when you began editing it.

Special Vi Commands

Material Covered:

:set	command: enables, disables, sets, or lists options:
autoindent	option: enables/disables automatic indentation:
autowrite	option: enables/disables automatic writing to the <i>vi</i> buffer after an editing session:
ignorecase	option: disables/enables upper- and lower-case distinction:
list	option: enables/disables tab and end-of-line markers:
magic	option: enables/disables extended set of metacharacters:
number	option: enables/disables line numbering:
shiftwidth	option: defines number of columns per shift-width:
showmatch	option: enables/disables parenthesis-, brace-, and bracket-matching:
slowopen	option: enables/disables screen refresh only when [ESC] is pressed:
wrapmargin	option: defines number of columns in right margin:
timeout	option: enables/disables one second time limit for macro entry:
readonly	option: enables/disables write protection for file:
paragraphs	option: defines the macro names recognized by the { and } commands:
sections	option: defines the macro names recognized by the [[and]] commands:
:map	command: defines macros:
:unmap	command: deletes macros:
ctrl-V	command: used to alter the meaning of special keys or characters:
:ab	command: defines abbreviations:
:una	command: deletes abbreviations:
:r	command: read contents of file or output of shell command into current file:
:w	command: write part or all of <i>vi</i> buffer to current file or to another file:
:e	command: edit same file over again, or begin editing another file:
:n	command: edit next file in <i>vi</i> argument list:
!	command: enables portions of the <i>vi</i> buffer to be filtered through an HP-UX command.

Setting Vi Options

Vi has several options that you can set for the duration of your editing session.

The **autoindent** option, when set, automatically indents each line of text so that it begins in the same column as the previous line. While inserting text, you cannot backspace over this indentation, but you can backtab over it with ctrl-D. This option is helpful when typing in program text. To enable this option, type

```
:set ai[RETURN]
```

Disable this option by typing

```
:set noai[RETURN]
```

The default is **noai**.

The **autowrite** option, when set, automatically writes the contents of the *vi* buffer to the current file you are editing when you quit editing the current file. This is helpful when you change files or leave the editor using commands that do not normally save the contents of the *vi* buffer. To enable this option, type

```
:set aw[RETURN]
```

Disable this option by typing

```
:set noaw[RETURN]
```

The default is **noaw**.

The **ignorecase** option, when set, causes *vi* to ignore case in searches. To enable this option, type

```
:set ic[RETURN]
```

Disable this option by typing

```
:set noic[RETURN]
```

The default is **noic**.

The **list** option, when set, causes a tab to be printed as "`^I`", and marks the end of each line with a dollar sign. To enable this option, type

```
:set list[RETURN]
```

Disable this option by typing

```
:set nolist[RETURN]
```

The default is **nolist**.

The **magic** option, when set, causes the period, left and right brackets, the asterisk, and the character combinations `\<` and `\>` to be treated in a special way when used in search patterns (see the section entitled *Searching for a Pattern*). To enable this option, type

```
:set magic[RETURN]
```

Disable this option by typing

```
:set nomagic[RETURN]
```

The default is **nomagic**.

The **number** option, when set, causes line numbers to be prefixed to each text line on your screen. To enable this option, type

```
:set nu[RETURN]
```

Disable this option by typing

```
:set nonu[RETURN]
```

The default is **nonu**.

The **shiftwidth** option enables you to specify the number of columns to skip when using <, <<, >, >>, ctrl-D, and ctrl-T (see the section entitled *Shifting Lines*). Ctrl-D backtabs over inserted shift-widths (using <, <<, >, or >>) or any indentation provided by the **autoindent** option. Ctrl-T inserts one shift-width at the beginning of the current line during text insertion. To set this option, type

```
:set sw = val[RETURN]
```

where *val* is the number of columns to skip. The default is **sw = 8**.

The **showmatch** option, when set, causes *vi* to show you the opening parenthesis, brace, or bracket when you type the corresponding closing parenthesis, brace, or bracket. This is helpful in complex mathematical expressions. To enable this option, type

```
:set sm[RETURN]
```

Disable this option by typing

```
:set nosm[RETURN]
```

The default is **nosm**.

The **slowopen** option, when set, causes *vi* to wait until you press [ESC] to update the screen after inserting or appending text. This is used on slow terminals to decrease the amount of time spent waiting for the screen to be updated. To enable this option, type

```
:set slow[RETURN]
```

Disable this option by typing

```
:set noslow[RETURN]
```

The default is **slow**.

The **wrapmargin** option enables you to specify the number of columns you want in your right margin. This is used when you are using continuous text input (see section entitled *Continuous Text Input*). To set this option, type

```
:set wm = val[RETURN]
```

where *val* is the number of columns in your right margin. The default is **wm = 0**.

The **timeout** option, when set, places a one second time limit on the amount of time it takes you to type in a macro name (see the section entitled *Defining Macros*). To enable this option, type

```
:set to[RETURN]
```

Disable this option by typing

```
:set noto[RETURN]
```

The default is **to**.

The **readonly** option, when set, places write protection on the file you are editing. This is used when you want simply to look at a file, and you want to ensure that you do not inadvertently change or destroy the contents of the file. To enable this option, type

```
:set readonly[RETURN]
```

Disable this option by typing

```
:set noreadonly[RETURN]
```

The default is **noreadonly**.

The **paragraphs** option contains the list of macro names recognized by the { and } commands as marking the beginning and end of a paragraph. Suppose you have three macros, .PG, .P, and .EP, that you want *vi* to recognize as paragraph delimiters. All you have to do is type

```
:set para = PGP EP[RETURN]
```

Note that, if a macro name is only one character long, you must type the single character macro name, followed by a space. The default **paragraph** string is

```
para = IPLPPPQPbpP LI
```

You may add your macros to this string, or completely redefine it using different macro names.

The **sections** option contains the list of macro names recognized by the `[[` and `]]` commands as marking the beginning and end of a section. **Sections** is defined in exactly the same way as **paragraphs** above. The default list of macro names is

```
sect = NHSHH HU
```

There are several other options available, but they are less commonly used than these. You can get a list of all possible options and their settings by typing

```
:set all[RETURN]
```

A list of all the options which you have changed is generated by typing

```
:set[RETURN]
```

If you want to know the value of a particular option, type

```
:set opt?[RETURN]
```

where *opt* is the name of the option. Note that multiple options can be set on one line, as in

```
:set ai aw nu[RETURN]
```

If a number argument is specified before the **:set** command, it is interpreted to be the new window size (number of lines per screenful of text).

Defining Macros

`Vi` has a macro facility which enables you to substitute a single keystroke for a longer sequence of keystrokes. If you are repeatedly typing the same sequence of commands, then you can probably save time and typing by defining a macro to perform the sequence of commands for you.

You use the **:map** command to define a macro. After the **:map** command, you type the key or keys that invoke the macro, and then the sequence of keystrokes that you want to put in the macro. For example,

```
:map d d4w[RETURN]
```


causes **d** to delete the next four words every time it is pressed. Also,

```
:map c /I ctrl-V[RETURN]dwiYou ctrl-V[ESC][RETURN]
```

causes **c** to find an occurrence of "I ", delete it, and replace it with "You ". The ctrl-V command tells *vi* to simply enter the next keystroke into the text of the macro and to ignore any special meaning that keystroke might have. The ctrl-V command is used above to flag [RETURN] and [ESC], both of which would have terminated the **:map** command before it was completed. Instead, [RETURN] and [ESC] are entered as keystrokes in the macro string. The final [RETURN] terminates the **:map** command.

If the macro name specified consists of a pound sign (#) followed by a number in the range 0 – 9, then a special function key on your terminal is mapped. For example,

```
:map #3 cILLUSTRATION GOES HEREctrl-V[ESC][RETURN]
```

maps special function key number 3 such that, when pressed, it changes the current line to the line "ILLUSTRATION GOES HERE". Of course, this feature is valid only on terminals which have special function keys.

Vi normally allows only one second to enter a macro name, so you should use only one keystroke to invoke the macro. However, if the **notimeout** option is set, *vi* imposes no time limit. If this is the case, you can use up to 10 keystrokes to invoke a macro. The sequence of keystrokes that define the macro can contain up to 100 keystrokes.

The **u** (undo) command, when invoked after a macro has been executed, reverses the effects of the entire macro.

Previously defined macros can be deleted with the **:unmap** command. For example, to delete the **c** macro defined above, type

```
:unmap c
```

If a number argument is specified before the **:map** or **:unmap** command, it is interpreted to be the new window size (number of lines per screenful of text).

Defining Abbreviations

You can define an abbreviation with the **:ab** command. For example,

```
:ab CRT cathode ray tube[RETURN]
```

defines "CRT" as an abbreviation that is expanded to "cathode ray tube" everywhere you type "CRT" in the text. Also,

```
:ab cs Department of Computer Sciences[RETURN]
```

defines "cs" as an abbreviation that is expanded to "Department of Computer Sciences" everywhere you type "cs" in the text.

The abbreviation name must contain only letters, digits, or underscores. *Vi* only expands abbreviations when they are delimited by white space on both sides, or by white space on the left and punctuation on the right. Abbreviations are not expanded if they appear as part of another word.

Abbreviations can be deleted with the **:una** command. For example,

```
:una cs
```

deletes the abbreviation associated with "cs".

If a number argument is specified before the **:ab** or **:una** command, it is interpreted to be the new window size (number of lines per screenful of text).

Reading Data Into Your Current File

The **:r** command enables you to read the contents of a file or the standard output from a shell command into the file you are currently editing. For example,

```
:r test_data[RETURN]
```

reads the contents of the file **test_data** into the current file after the cursor. Also,

```
:7r std_dev[RETURN]
```

reads the contents of the file **std_dev** into the current file after line seven.

You can also read the output from a shell command into your file by typing

```
:r !cmd[RETURN]
```

where *cmd* is the name of the shell command. For example,

```
:r !ls[RETURN]
```

reads a list of the files in your working directory into the file you are editing, beginning at the current cursor position.

If a number argument is specified before the `:r` command, it is interpreted to be the new window size (number of lines per screenful of text).

Writing Edited Text Onto a File

The `:w` command is used to write the current contents of the *vi* buffer onto a file. The contents of the *vi* buffer remain unchanged. It is a good idea to write the contents of the *vi* buffer onto a file periodically, especially if you have been editing the file for a long time, and have made significant changes. That way, should a system crash or a power failure occur, some or all of your changes are saved.

If you specified a file name when you invoked *vi*, then you need not specify a file name if you want to write to the current file. *Vi* remembers the name of the file you are editing or creating, and writes to that file by default. For example, if you invoked *vi* as

```
$ vi test_data
```

then you need only type

```
:w[RETURN]
```

to write the contents of the *vi* buffer onto **test_data**. However, if you did not specify a file name when you invoked *vi*, then you must supply a file name with the `:w` command. For example, if you invoked *vi* as

```
$ vi
```

then you must type

```
:w filename[RETURN]
```

where *filename* is the name of the file on which you want the contents of the *vi* buffer to be written.

You can write your changes to an existing file other than the one you are editing. For example,

```
:w! format[RETURN]
```

writes your changes to the file **format**. Note that the exclamation point tells *vi* to overwrite the previous contents of **format** with the contents of the *vi* buffer.

You can also write your changes to a file that does not yet exist. For example,

```
:w thesis[RETURN]
```

causes *vi* to create a file called **thesis**, and writes all your changes on **thesis**.

You can specify that a portion of your text be written to another file that does not yet exist. For example,

```
:2,35w prog[RETURN]
```

creates a file called **prog** and writes line 2 through line 35 of the current file on **prog**. The same thing can be done with a file that already exists, as in

```
:3,10w! list[RETURN]
```

writes line 3 through line 10 of the current file on the file called **list**. The exclamation point causes the previous contents of **list** to be destroyed and replaced by the specified portion of the *vi* buffer.

If a number argument is specified before the **:w** command, it is interpreted to be the new window size (number of lines per screenful of text).

Note that, while you may append other files to the file you are currently editing, *vi* provides no facilities that enable you to append the current file to another file.

Editing Other Files

The **:e** command enables you to edit other files without leaving *vi*. For example,

```
:e report[RETURN]
```

tells *vi* to stop editing the current file and to start editing **report**. If **report** does not exist, *vi* creates it for you. Note that *vi* requires a **:w** command to precede a **:e** command, so that the previous contents of the *vi* buffer are saved (unless the **autowrite** option is set, in which case *vi* is silent).

You can also tell *vi* to start editing a file beginning with a particular line. For example,

```
:e + test[RETURN]
```

tells *vi* to start editing **test**, beginning with the last line of the file. Also,

```
:e + M letter[RETURN]
```

tells *vi* to start editing *letter* at the middle line of the screen. Any *vi* command discussed in the section entitled *Moving Around in the File* and not containing any spaces can be inserted after the "+" in the previous examples. For example,

```
:e + CAE/+0ctrl-V[RETURN] cov_let[RETURN]
```

tells *vi* to start editing `cov_jet`, with the cursor positioned at the beginning of the first line containing the string "CAE". Note that `ctrl-V` had to be used to flag [RETURN] so that the `:e` command is not terminated before it is completed.

If you decide that you do not like the changes you have made to a file, you can discard the changes and begin editing the same file over again by typing

```
:e![RETURN]
```

The exclamation point tells *vi* that you know what you are doing, and that you do not want to save the current contents of the *vi* buffer. To discard the changes and begin editing a different file, type

```
:e! name[RETURN]
```

where *name* is the name of the file you want to edit. Again, the exclamation point tells *vi* that a `:w` command is not necessary.

If a number argument is specified before the `:e` command, it is interpreted to be the new window size (number of lines per screenful of text).

Editing the Next File in the Argument List

The `:n` command tells *vi* to stop editing the current file and begin editing the next file in the argument list. For example,

```
:n[RETURN]
```

tells *vi* to start editing the next file in the argument list. *Vi* insists that you use a `:w` command before you begin editing the next file, unless you type

```
:n![RETURN]
```

which tells *vi* to discard any changes you have made to the current file, and begin editing the next file.

If a number argument is specified before the `:n` command, it is interpreted to be the new window size (number of lines per screenful of text).

Filtering Buffer Text Through HP-UX Commands

Portions of the *vi* buffer text can be given as input to an HP-UX command, the output of which is then re-inserted into the previous location of that text. The **!** command is used to invoke filtering.

For example, suppose you have a list of items, one per line, that you want to sort alphabetically. This is easily done in several ways. If a single **!** is used, then you must supply modifiers which specify the extent of the text to be sorted. Let's assume that your file looks like this:

```
.PP
crackers
peas
roast
apples
oranges
tomatoes
grapes
.PP
```

.PP is an *nroff*/*troff* paragraph macro, which is recognized by **{** and **}** as beginning and ending a paragraph. Thus, if your cursor is positioned at the beginning of the first **.PP** macro, and you type

```
!)sort[RETURN]
```

then the list of grocery items is replaced by the output from the *sort* command. The **}** command is used to select the next paragraph as input for *sort*.

A second way to sort the same text is by typing

```
7!!sort[RETURN]
```

If two **!**'s are typed, then whole lines are assumed, and the number argument specifies how many whole lines to sort. For this example to work, your cursor must be somewhere on the "crackers" line.

Note that, in both of the above examples, a single **!** and the command name is all that is printed at the bottom of your screen. No number arguments or modifiers are echoed.

Any HP-UX command with useful output can be used in place of *sort*, depending on what you want to do. Since *vi* has no right margin justification function, another useful command might be *nroff*, which could be used to justify right margins or perform other formatting.

Note that filtering affects only the buffer contents, not the actual contents of your current file.

Vi and Ex

Material Covered:

Q command: escape from *vi* into *ex*;
vi command: escape from *ex* back to *vi*.

Vi is actually one mode of editing within the editor *ex*. In fact, all of the commands beginning with **:** are also available in *ex*. You can escape to the *ex* line-oriented editor by giving the command **Q**. When the **Q** command is given, *vi* responds with a line of information, and then *ex* takes over and prints the *ex* prompt (**:**). To get from *ex* to *vi*, type **vi** after the *ex* prompt. *Vi* clears the screen and prints a screenful of text, with your current line at the time you typed **vi** at the top of the screen.

There are several things which can be done more easily in *ex*, the most notable of which are global searches and substitutions. Thus, you may find yourself, after a while, switching between the two editing modes to access functions which are better handled by one or the other. For information concerning the *ex* editor, refer to the *Ex Reference Manual* included in *HP-UX Selected Articles*.

The Shell Interface

Material Covered:

vi command: invokes the *vi* editor;
view command: invokes the *vi* editor in read-only mode;
!: command: escape to the shell for the duration of one command;
:sh command: escape to the shell indefinitely;
ZZ command: writes the contents of *vi* buffer to current file and leaves editor;
:q! command: discards contents of *vi* buffer and leaves editor.

Getting Into Vi

There are two ways to invoke *vi* from the shell, one of which is to type

```
$ vi
```

optionally followed by the names of the files you want to edit. You can also invoke *vi* by typing

```
$ view
```

optionally followed by the names of the files you want to edit. *View* is the same editor as *vi*, except that the **readonly** option is automatically set. This protects the contents of a file from being accidentally overwritten or destroyed. *View* is used whenever you want to look at an important file, but you do not want to change its contents.

Note that the **readonly** option can be disabled or overridden while you are in *view*. Nothing prevents you from typing

```
:set noreadonly[RETURN]
```

which simply changes *view* into *vi*. Also, you can still overwrite the contents of a file when the **readonly** option is set by using a **:w!** command.

Getting Back to the Shell

You can get back to the shell temporarily in either of two ways. You can execute a shell command while editing a file by typing

```
!:cmd[RETURN]
```

where *cmd* is the name of the shell command you want to execute. For example,

```
!:ls[RETURN]
```

prints a list of all the files in your working directory. Once the command has been executed, you can either enter another command with **!:**, or you can continue editing where you left off by pressing [RETURN]. If you press [RETURN], *vi* responds by clearing the screen and displaying the text you were working on before the shell command was executed.

You can escape to a shell temporarily by typing

```
:sh[RETURN]
```

This puts you in a shell, where you can execute as many commands as you want. When you want to continue editing, press ctrl-D. *Vi* clears the screen and displays the text you were working on.

If a number argument is specified before the **!:** or **:sh** command, it is interpreted to be the new window size (number of lines per screenful of text).

There are two ways to return to the shell permanently. If you want to save all your changes to the current file and return to the shell, use the **ZZ** command. **ZZ** writes the contents of the *vi* buffer onto the current file (if any changes have been made), and leaves the editor.

If you do not want to save the changes you have made to the current file, then use **:q!**. **:q!** simply leaves the editor and discards the contents of the *vi* buffer. The file you were editing is left unchanged. You should be very sure that this is what you want to do, since the contents of the *vi* buffer are permanently lost.

If a number argument is specified before the **:q!** command, it is interpreted to be the new window size (number of lines per screenful of text).

Miscellaneous Topics

Material Covered:

.profile	file; automatically executed by the shell at login; can contain macros, abbreviations, and option settings; must reside in your home directory;
EXINIT	variable; placed in .profile file; contains macro, abbreviation, and option information;
.exrc	file; contains <i>ex</i> and <i>vi</i> initialization constructs; this file is automatically scanned by <i>ex</i> if EXINIT is not defined;
buffers 1–9	buffers; contain the last nine text deletions performed during the current edit session;
ctrl-V	operator; enable control characters to be inserted in file;
z	command; adjust and redefine window size;
ctrl-L	command; refreshes the screen;
ctrl-G, :f	command; provide information about your current edit session.

Vi Initialization

Option settings, macros, and abbreviations last only the length of your editing session, after which they either return to default settings or become undefined. If you do not want to bother with resetting these things each time you invoke *vi*, you can put your option settings, macros, and abbreviations in a file called **.profile**. This file is automatically executed by the shell when you log in. The **.profile** file must reside in your home directory.

If you include *vi* information in your **.profile**, they must be placed in a string and set equal to the variable EXINIT. EXINIT is a variable that is assumed by the system to contain information pertinent to the *vi* editor. For example, to set the **autoindent**, **autowrite**, and **number** options and define two macros, put the following two statements in your **.profile** file in your home directory:

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

This EXINIT string sets the **autoindent**, **autowrite**, and **number** options and defines the two macros (@ and #, which delete one line and one character, respectively. Note that each **set** and **map** is separated from the next by a vertical bar (|), and that the entire string is enclosed in single quotes and set equal to EXINIT. The *export* command makes the information in EXINIT available to all processes you create.

If EXINIT is not defined when *vi* is invoked, then *vi* looks for the file **.exrc** in your home directory. If it is found, *vi* scans its contents, assuming that the information contained therein consists of various commands for setting up mapping, abbreviations, options, etc.

If the amount of initialization for *vi* is extensive, it is usually more convenient to forget about EXINIT, and use the **.exrc** file instead. To use EXINIT, the information must be specified in a string enclosed in single quotes. This could prove to be a very long string if there is a lot of initialization to do. Strings this size are normally hard to read and hard to input.

Appendix B at the end of this article contains a listing of the default **.exrc** file shipped with your system. You are free to use this file as your own **.exrc** file if you wish. To do so, simply copy the file */etc/d.exrc* into your home directory, and rename it **.exrc**. Your system administrator may have already done this for you. To find out, list the files in your home directory using **ls -a**.

Recovering Lost Lines

Vi has nine buffers, numbered 1 through 9, in which the last nine text deletions are automatically stored. Thus, you can specify one of these buffers with the **p** or **P** command to recover a deletion. For example, "**3p**" puts the deleted text stored in buffer 3 into the *vi* buffer after or below the cursor.

The **.** command, which repeats the last command that made a change in your text, automatically increments the buffer number if the last command referenced a numbered buffer. Thus, "**1p.....**" prints out all the text deleted in the last nine deletions. If you want to put a particular block of deleted text back into your file, but you do not know which buffer to look in, you can perform a sequence of commands like "**1pu.u.u.**" (and so on), which prints the contents of each buffer until you find the text you want. The **u** command gets rid of the unwanted text you encounter as you search.

Note that text stored in buffers 1 through 9 is preserved between files (as long as you do not exit the editor itself), so you may insert deleted text from one file into another by using buffers 1–9 and the **p** or **P** command.

Entering Control Characters in Your Text

If you need to put a control character in your text, you must precede the control character with a **ctrl-V**. The **ctrl-V** causes a caret (^) to be printed on your screen, showing that the next character is to be interpreted as a control character. For example, to enter a **ctrl-L** in your text, type

```
ctrl-V ctrl-L
```

This causes two characters, "**^L**", to be printed on your screen. If you try to backspace over them, however, you can see that they are actually one character.

You may enter any control character into your file except one: the null character (**ctrl-@**). There is also a restriction that applies to the line-feed character (**ctrl-J**). A linefeed is not allowed to occur anywhere except the beginning of a line, because *vi* uses the linefeed to separate lines in your file.

Adjusting the Screen

If a transmission error or the output from a program causes your screen to become cluttered, you can refresh the screen by pressing **ctrl-L**. *Vi* clears the screen and reprints the text you were working on.

The **z** command is used to position specific lines on the screen. **z[RETURN]** places the current line at the top of the window, **z.** places the current line in the middle of the window, and **z-** places the current line at the bottom of the window. If a number argument *n* is specified after **z** but before the modifier, then the window size is changed to be *n* lines long after **z** has executed. If *n* is specified before **z**, then **z** places line number *n* (instead of the current line) at the top, middle, or bottom of the new screen. For example, **z10-** places the current line at the bottom of a 10-line window. Also, **6z.** places line number 6 at the middle of the screen, leaving the window size unchanged.

Printing Your File Status

If you are editing a file and lose track of where you are in the file, or if you forget the name of the file you are editing, the **ctrl-G** command can help you. In response to the **ctrl-G** command, *vi* prints the name of the file you are editing, the number of the current line, the number of lines in the buffer, and how much of the buffer you have already edited (expressed as a percentage). The **:f** command is a synonym for **ctrl-G**.

Appendix A: Character Functions

This appendix gives the *vi* meanings associated with each character in the ASCII character set. The characters are presented in the following order: control characters, special characters, digits, upper-case characters, and lower-case characters. For each character, its meaning is given as a command and during an insert, as appropriate. (Note that the control key (CTRL) is represented by `^` in the following list).

- `^@` Not a command character. If it is typed as the first character of an insertion, it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted, the mechanism is not available. A `^@` cannot be part of the file text due to the editor's implementation.
- `^A` Unused.
- `^B` Moves backward one page. A preceding integer specifies the number of pages to move over. Two lines of continuity are kept if possible.
- `^C` Unused.
- `^D` As a command, scrolls forward one half of a page. A preceding integer specifies the number of logical lines to scroll for each command. This integer is remembered for all future `^D` and `^U` commands. During an insert, `^D` backtabs over **autoindent** white space inserted at the beginning of a line. This white space cannot be backspaced over.
- `^E` Exposes one more line at the bottom of the screen, leaving the cursor at its present position, if possible.
- `^F` Moves forward one page. A preceding integer specifies the number of pages to move over. Two lines of continuity are kept if possible.
- `^G` Prints the name of the current file, whether it has been modified, the current line number, the number of lines in the file, and how much of the buffer you have already edited (expressed as a percentage).
- `^H (BS)` Same as **left arrow** (see **h**). During an insert, eliminates the last input character, backing over it but not erasing it. The character remains so you can see what you typed if you wish to type something only slightly different.
- `^I (TAB)` Not a command character. When inserted, it prints as some number of spaces. When the cursor is at a tab character, it rests at the last of the spaces which represent the tab. The spacing of tabstops is controlled by the **tabstop** option.
- `^J (LF)` Same as **down arrow** (see **j**).
- `^K` Unused.
- `^L (FF)` Causes the screen to be cleared and redrawn.
- `^M (CR)` Advances to the next line, at the first printable character on the line. If preceded by an integer, *vi* advances that many lines. During an insert, `^M` causes the insert to continue onto another line.
- `^N` Same as **down arrow** (see **j**).
- `^O` Unused.
- `^P` Same as **up arrow** (see **k**).
- `^Q` Not a command character. In input mode, `^Q` quote the next character, the same as `^V`, except that some teletype drivers do not allow `^Q` to be seen by *vi*. Use `^V` instead.
- `^R` Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single `^@` character on them). On hardcopy terminals in *open* mode, `^R` retypes the current line.

<code>^S</code>	Unused.
<code>^T</code>	Not a command character. During an insert, with autoindent set and at the beginning of the line, inserts shiftwidth white space.
<code>^U</code>	Scrolls up one page. A preceding integer specifies the number of lines to scroll. This integer is remembered for all future <code>^D</code> and <code>^U</code> commands. On a dumb terminal, <code>^U</code> will clear the screen and redraw it further back in the file.
<code>^V</code>	Not a command character. In input mode, <code>^V</code> quotes the next character so that it is possible to insert non-printing and special characters into the file, and include special characters in macros, abbreviations, etc.
<code>^W</code>	Not a command character. During an insert, <code>^W</code> mimics a b command, thus deleting all inserted characters from the current cursor location to the beginning of the previous word. The deleted characters remain on display. (See <code>^H</code>).
<code>^X</code>	Unused.
<code>^Y</code>	Exposes one more line at the top of the screen, leaving the cursor in its present position, if possible.
<code>^Z</code>	Unused.
<code>^[(ESC)</code>	Cancels a partially formed command, such as a z command when no following character has yet been given. It also terminates inputs on the last line (read by commands such as <code>:</code> , <code>/</code> , and <code>?</code>), and ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. You can thus press ESC if you don't know what is happening until the editor rings the bell.
<code>^\</code>	Unused.
<code>]]</code>	Searches for the word which immediately follows the cursor. It is equivalent to typing the <i>ex</i> command <code>:ta</code> , followed by that word, followed by RETURN.
<code>^^</code>	(Control- <code>^</code>), equivalent to the <i>ex</i> command <code>:e #</code> , which returns you to the previous position in the last edited file, or edits a file you specified if you got a "No write since last change" diagnostic, and you don't want to type the file name again. (In the latter case, you will have to do a <code>:w</code> before <code>^^</code> will work. If you don't want to write the file, then do a <code>:e! #</code> instead).
<code>_</code>	Unused.
SPACE	Same as right arrow (see I).
<code>!</code>	An operator which processes lines from the buffer with reformatting commands. Follow <code>!</code> with the object to be processed, and then the command name terminated by RETURN. Doubling <code>!</code> and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the <code>!</code> . Thus, <code>2! sort</code> , followed by RETURN, sorts the next two paragraphs by running them through the <i>sort</i> command. To read a file or the output of a command into the buffer use <code>:r</code> . To simply execute a command use <code>!:</code> .
<code>"</code>	Precedes a named buffer specification. There are named buffers 1 – 9 used for saving deleted text, and named buffers a – z into which text can be placed.
<code>#</code>	The macro character which, when followed by a number, will substitute for a function key on terminals without function keys. In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a <code>\</code> to insert it, since it normally backs over the last input character you gave.

- \$ Moves to the end of the current line. If you execute **:set list**, then the end of each line will be shown by printing a \$ after the end of the displayed text in the line. Given a count, \$ advances to the end of the line that many lines from the current line (i.e. **3\$** advances to the end of the line two lines after the current line).
- % Moves to the parenthesis or brace which balances the parenthesis or brace at the current cursor position.
- & A synonym for the *ex* command, **:&**.
- ' When followed by another ', returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a – z**, returns to the line which was marked with this letter with the **m** command, at the first non-space character in the line. When used with an operator, such as **d**, the operation takes place over complete lines.
- (Moves to the beginning of a sentence, or to the beginning of a LISP s-expression if the **lisp** option is set. A sentence ends at a **, !, or ?** which is followed by either the end of a line or by two spaces. Any number of closing **),], ", and '** characters may appear after the **, !, or ?**, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries. A count advances that many sentences.
-) Advances to the beginning of a sentence. A count repeats the effect. See the description of **(** above for a description of a sentence.
- * Unused.
- + Same as carriage-return when used as a command.
- . Reverses the last **f, F, t, or T** command, looking the other way in the current line. A count repeats the search.
- Moves to the previous line at the first non-white-space character. This is the inverse of **+** and **RETURN**. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if this is not possible. If a large amount of scrolling would be required, the screen is also cleared and redrawn, with the current line at the center.
- . Repeats the last command which changed the *vi* buffer. Especially useful when deleting words or lines: you can delete some words/lines and then hit **.** to delete more words/lines. Given a count, it passes it on to the command being repeated.
- / Used to initiate a forward search for a pattern. If you press **/** accidentally, you can use **BACKSPACE** to return to your previous position.
- 0 Moves to the first character on the current line. Also used to form numbers after an initial **1 – 9**.
- 1 – 9 Used to form numeric arguments to commands.
- : A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with **RETURN**, and the command is then executed. You can return to your previous position by pressing **DEL** or **RUB** if you press **:** accidentally.
- : Repeats the last single character search using **f, F, t, or T**. A count iterates the basic scan.
- < An operator which shifts lines left one **shiftwidth**, normally 8 spaces. Like all operators, **<** affects lines when repeated, as in **<<**. Counts cause **<** to act on more than one line.
- = Re-indent a line for LISP, as though the line was typed in with the **lisp** and **autoindent** options set.

- > An operator which shifts lines right one **shiftwidth**, normally 8 spaces. Affects lines when repeated, as in >>. A count causes > to act on more than one line.
- ? Initiates a backwards search for a pattern. If you press / accidentally, you can use BACKSPACE to return to your previous position.
- @ A macro character. If this is your kill character, you must escape it with a \ to type it in during input mode, as it normally backs over the input you have given on the current line.
- A Appends at the end of a line, a synonym for \$a.
- B Backs up one word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the command.
- C Changes the rest of the text on the current line; a synonym for c\$.
- D Deletes the rest of the text on the current line; a synonym for d\$.
- E Moves forward to the end of a word. A count repeats the command.
- F Finds a single following character, backwards in the current line. A count repeats the search.
- G Moves to the line number given as a previous argument, or the end of the file if no preceding argument is given. The screen is redrawn with the new current line in the center if necessary.
- H Homes the cursor to the top line of the screen. If a count is given, the cursor is moved to the count-th line on the screen. In all cases, the cursor is moved to the first non-white-space character on the line.
- I Inserts at the beginning of a line; a synonym for ^i.
- J Joins lines together, supplying appropriate white space: one space between words, two spaces after a ., and no spaces at all if the first character of the line to be appended is). A count causes that many lines to be joined rather than the default two.
- K Unused.
- L Moves the cursor to the first non-white-space character of the last line on the screen. If a count is given, the cursor is moved to the first non-white-space character of the count-th line from the bottom.
- M Moves the cursor to the middle line on the screen, at the first non-space character.
- N Scans for the next match of the last pattern given to / or ?. but in the opposite direction.
- O Opens a new line above the current line and inputs text there, up to an ESC.
- P Puts the last deleted text back before or above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise, the text is inserted before the current location of the cursor. May be preceded by a buffer name to retrieve the contents of that buffer.
- Q Quits from *vi* and goes to *ex* mode. In this mode, whole lines form commands, ending with RETURN. All : commands can be given; the editor supplies the : prompt.
- R Replaces characters on the screen with characters you type (overlay fashion). Terminates with ESC.
- S Changes whole lines; a synonym for cc. A count substitutes for that many lines. The lines are saved in numeric buffers, and erased on the screen before the substitution begins.
- T Takes a single following letter, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the command that many times.

U	Restores the current line to its state before you started changing it.
V	Unused.
W	Moves forward to the beginning of a word in the current line, where words are defined as sequences of non-space characters. A count repeats the command.
X	Deletes the character before the cursor. A count repeats the command, but only characters on the current line are deleted.
Y	Yanks a copy of the current line into the unnamed buffer, to be put back by a later p or P . A count yanks that many lines. Can be preceded by a buffer name to put text into that buffer.
ZZ	Exits the editor (same as :x). If any changes have been made, the buffer is written out to the current file, and the editor terminates.
[[Backs up to the previous section boundary, which is marked by a particular macro invocation (the names of which are specified in the sections option), or by ^L (formfeed). Lines beginning with { also stop [[, making it useful for looking backwards through C programs. If the lisp option is set, [[also stops at each (it finds at the beginning of a line.
\	Unused.
]]	Moves forward to a section boundary (see description of [[).
^	Moves to the first non-space character on the current line.
_	(Underscore) Unused.
`	When followed by a ` , returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a lower-case letter, returns to the position which was marked with this letter with an m command. When used with an operator such as d , the operation takes place from the exact marked place to the current position within the line: if you use ` , the operation takes place over complete lines.
a	Appends arbitrary text after the current cursor position. The insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with ESC.
b	Backs up to the beginning of a word in the current line. A word is a sequence of alphanumeric, or a sequence of special characters. A count repeats the command.
c	An operator which changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text which is being changed is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed away is marked with a \$. A count causes that many objects to be changed.
d	An operator which deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected.
e	Advances to the end of the next word. A count repeats the command.
f	Finds the first instance of the next character following the cursor on the current line. A count repeats the command.
g	Unused.
h	Left arrow. Moves the cursor one character to the left. Like the other arrow keys, either h , the left arrow key, or one of the synonyms (^H) has the same effect. A count repeats the command.

i	Inserts text before the cursor. Otherwise, i is like a .
j	Down arrow. Moves the cursor down one line in the same column. If the position does not exist, <i>vi</i> comes as close as possible to the same column. Synonyms include ^J (linefeed) and ^N.
k	Up arrow. Moves the cursor up one line in the same column. Synonym is ^P.
l	Right arrow. Moves the cursor one character to the right. SPACE is a synonym.
m	Marks the current position of the cursor in the mark register which is specified by the next character (a – z). Return to this position or use with an operator by preceding the mark letter with ` or `.
n	Repeats the last search specified with / or ?.
o	Opens a new line below the current line. Otherwise, o is like O .
p	Puts text after/below the cursor. Otherwise, p is like P .
q	Unused.
r	Replaces the single character marked by the cursor with a single character you type. The new character may be a RETURN (this is the easiest way to split lines). A count <i>n</i> replaces the next <i>n</i> characters with the character you type.
s	Changes the single character marked by the cursor to the text which follows, up to an ESC. Given a count, that many characters are replaced by the text. The last character to be changed is marked with a \$.
t	Advances the cursor up to the character before the next character typed on the current line. A count repeats the command.
u	Undoes the last change made to the current buffer. If repeated, will alternate between these two states. It is thus its own inverse. When used after an insert which inserted text on more than one line, the lines are saved in the numeric buffers.
v	Unused.
w	Advances to the beginning of the next word. A count repeats the command.
x	Deletes the single character marked by the cursor. A count causes that many characters to be deleted. Works only on the current line.
y	An operator which yanks the following object into the unnamed temporary buffer. If preceded by a buffer name, the text is placed in that buffer also. Text can be recovered with a later p or P .
z	Redraws the screen with the current line placed as specified by the following character: RETURN specifies the top of the screen, . specifies the center of the screen, and – specifies the bottom of the screen. A count may be given after z and before the following character to specify the new window size for the redraw. A count before z gives the number of the line to place in the center of the screen instead of the current line.
{	Moves to the beginning of the preceding paragraph. A paragraph begins at a macro invocation defined in the paragraphs option, and at the beginning of a section. A paragraph also starts at a blank line.
	Places the cursor on the character in the column specified by the count.
}	Advances to the beginning of the next paragraph. See { for the definition of a paragraph.
~	Unused.
^?(DEL)	Interrupts the editor, returning it to command mode.

Appendix B: Example .exrc File

The following is a reproduction of the default `.exrc` file shipped with your system. It is useful as an example of how it can be used to set up certain *vi* and *ex* parameters prior to your editing session. These contents can be changed at any time should the need arise to customize the editors for a particular application. Also, note that the line numbers in the following listing do not appear in the file, but are included to clarify the explanatory material that follows.

```
1  set autoindent autowrite showmatch wrapmargin=0 report=1
2  map ^W :set wrapmargin=8^M
3  map ^Z {!}sort -b^M
4  map ^X {!}sort -b^M
5  map ^[h 1G
6  map ^[H 1G
7  map ^[F G
8  map ^[V ^B
9  map ^[U ^F
10 map ^[T ^Yk
11 map ^[S ^Ej
12 map ^[Q i
13 map ^[P x
14 map ^[L O
15 map ^[M dd
16 map ^[K D
17 map ^[J DjdG$
18 map! ^[A ^V
19 map! ^[D ^H
20 map! ^[C ^V
21 map! ^[B ^M
22 map! ^[L ^M
23 map! ^[Q ^[
24 map! ^[R ^[
```

In the above, the `^` character indicates that the CTRL (control) key is held down while the next following key is pressed. The `^[` sequence is the escape sequence, and is equivalent to the ESC key (if any) on your terminal. Here is a line-by-line description of the contents of the default `.exrc` file:

LINE ACTION

- 1 enables the **autoindent**, **autowrite**, and **showmatch** options, sets the **wrapmargin** option to 0, and sets the **report** option to one line.
- 2 maps the control-W sequence to the *ex* command:

```
:set wrapmargin = 8
```

The control-M at the end of the sequence is a carriage-return. This is entered into the *.exrc* file by pressing control-V followed by a carriage-return.

- 3 maps the control-Z sequence to a shell escape sequence. This sequence pipes the data from the beginning of the current line to the end of the current paragraph into the *sort(1)* command.
- 4 maps the control-X sequence to a shell escape sequence. This sequence pipes the data from the beginning of the current paragraph to the end of the current paragraph into the *sort(1)* command.
- 5 maps escape-h, a sequence often transmitted by the HOME key, to the editor command **1G** (go to line one of the file). This enables you to use the HOME key while editing in *vi*.
- 6 performs the same function as line 5.
- 7 maps escape-F, the sequence transmitted by the HOME DOWN key, to the editor command **G** (go to the last line of the file). This enables you to use the HOME DOWN key while editing in *vi*.
- 8 maps escape-V, the sequence transmitted by the PREV PAGE key, to the editor command **^B** (go back one page). This enables you to use the PREV PAGE key while editing.
- 9 maps escape-U, the sequence transmitted by the NEXT PAGE key, to the editor command **^F** (go forward one page).
- 10 maps escape-T, the sequence transmitted by the ROLL DOWN key, to the editor commands **^Yk** (scroll up one line; move cursor down one line).
- 11 maps escape-S, the sequence transmitted by the ROLL UP key, to the editor commands **^Ej** (scroll up one line; move cursor down one line).
- 12 maps escape-Q, the sequence transmitted by the INSERT CHAR key, to the editor command **i** (start insert mode).
- 13 maps escape-P, the sequence transmitted by the DELETE CHAR key, to the editor command **x** (delete current character).
- 14 maps escape-L, the sequence transmitted by the INSERT LINE key, to the editor command **O** (create a new line above the current line, and start insert mode).
- 15 maps escape-M, the sequence transmitted by the DELETE LINE key, to the editor command **dd** (delete current line).
- 16 maps escape-K, the sequence transmitted by the CLR LINE key, to the editor command **D** (delete to the end of the current line).

- 17 maps escape-J, the sequence transmitted by the CLR DISPLAY key, to the editor commands DjdG\$ (delete to end of line, go down one line, delete to end of file).
- 18 maps escape-A, the sequence transmitted by the UP ARROW key, to the sequence ^V (causes cursor to move one space to the right) when it is used in insert mode (**map!** causes a key to be defined in insert mode only).
- 19 maps escape-D, the sequence transmitted by the LEFT ARROW key, to the sequence ^H (causes cursor to move one space to the left) when it is used in insert mode.
- 20 maps escape-C, the sequence transmitted by the RIGHT ARROW key, to the sequence ^V (causes cursor to move one space to the right) when it is used in insert mode.
- 21 maps escape-B, the sequence transmitted by the DOWN ARROW key, to the sequence ^M (carriage-return) when it is used in insert mode.
- 22 maps escape-L, the sequence transmitted by the INS LINE key, to the sequence ^M (carriage-return) when it is used in insert mode. This makes the INS LINE key have the same definition in *vi* as it has in REMOTE mode.
- 23 maps escape-Q, the sequence often transmitted by the INS CHAR key, to the escape key during insert mode.
- 24 maps escape-R, the sequence often transmitted by the INS CHAR key, to the escape key during insert mode.

Table of Contents

The Ed Editor

Creating an Ordinary File	1
Getting Acquainted with Ed	2
Invoking Ed	2
Prompting	3
Error Messages	3
Moving Around in the File	3
Line Pointers	4
Pointer to the Current Line	4
Pointer to the Last Line	6
Setting Pointers to Lines	7
Searching for Strings	8
Forward Searches	8
Backward Searches	9
Repeating a Search	9
Line Number Arithmetic with Searches	9
Using Metacharacters With Searches	10
Adding, Deleting, and Correcting Text	12
Printing Lines	13
Appending Text	14
Inserting Text	15
Deleting Text	15
Undoing Commands	16
Changing Lines	16
Moving Lines	17
Copying Lines	18
Modifying Text Within a Line	19
Making Commands Effective Globally	22
Joining Lines Together	25
Splitting Lines Apart	25
Special Ed Commands	26
Finding the Currently Remembered File Name	26
Writing Buffer Text Onto a File	27
Reading Files Into the Buffer	28
Editing Other Files	29
Silencing the Character Counts	30
Encrypting and Decrypting Text	31
The Shell Interface	33
Escaping to the Shell Temporarily	33
Exiting the Editor	34
Miscellaneous Topics	35
Interrupting the Editor	35
Editing Scripts	35



The Ed Editor

Ed is an interactive, line-oriented text editor. Its purpose is to enable you to create ordinary files, and to add to, delete, or modify the text in those files.

Creating an Ordinary File

The remainder of this chapter contains several examples illustrating *ed* commands. These examples are more instructive if you try each of them on some text of your own. Thus, create an ordinary file by typing in the commands and text shown below in **bold** (portions of the example text shown below are taken from *A User Guide to the UNIX System*, by Rebecca Thomas and Jean Yates).

```
$ ed testfile
```

```
?testfile
```

```
a
```

```
The ed editor operates in two modes: command mode and  
text entry mode. In command mode, the edytor interprets  
your unput as a command. In text entry mode, ed adds  
your input to the text located in a special buffer where  
ed keeps a copy of the file you are editing. It is \ \ *.  
important to note that ed always makes changes to the  
copy of yourrr file in the buffer. The contents of the  
original file are not changed until you write the changes  
to the file.
```

```
.
```

```
w
```

```
461
```

```
q
```

```
$
```

Be sure to type in the text exactly as it is shown above. The mistakes are corrected later in the examples.

Getting Acquainted with Ed

Material Covered:

ed	command; invokes <i>ed</i> without a file name argument:
ed file	command; invokes <i>ed</i> with a file name argument:
P	command; enables/disables <i>ed</i> prompt (*);
h	command; explains the last question mark given by <i>ed</i> ;
H	command; enables/disables verbose error messages; explains the last question mark given by <i>ed</i> , and all future question marks.

Invoking Ed

Ed can be invoked in one of two ways. The first is to simply type **ed**, followed by [RETURN]. For example,

```
$ ed
```

invokes *ed* without a file name argument. When invoking *ed* this way, you must specify the file you want to edit with a separate command. It is more common to invoke *ed* by typing

```
$ ed filename
```

where *filename* is the name of the file you want to edit. This combines the two separate commands mentioned above into a single command.

Ed responds differently depending on whether or not the file already exists. Try creating a new file called **newfile**:

```
$ ed newfile
?newfile
```

Ed responds with "?newfile", which means that *ed* cannot find that file in your working directory. This is to be expected, since the file does not yet exist. *Ed* is now waiting for your commands to create and edit **newfile**.

If the file already exists, *ed* reads its contents into a buffer named `/tmp/e#`, where `#` is the number of the process running *ed*. *Ed* then displays a count of the characters contained in that file. You have a file called `testfile` in your working directory. You are probably still in *ed* from the previous example, so type `q[RETURN]` to exit *ed*, then edit `testfile` by typing

```
$ ed testfile
461
```

Ed tells you that `testfile` currently contains 461 characters. Do not exit *ed* this time, but leave it in its current state. The examples that follow pick up where you left off above.

Prompting

One of the most noticeable features of *ed* is its lack of prompts. When you type in a command, *ed* attempts to execute it, and, if successful, *ed* returns silently to you for another command. If an error is encountered, or a command cannot be executed for some reason, *ed* prints a question mark, and then silently waits for you to figure out the problem.

Many people find this silence desirable, but for those who do not, there are commands that make *ed* more friendly. The **P** command causes *ed* to prompt you with an asterisk (*). Executing the **P** command again turns off the prompt. By default, *ed*'s prompt is disabled.

Error Messages

As mentioned above, *ed*'s default error message is a single question mark (?). As you gain experience with *ed*, these question marks become easier to interpret, but for the beginning user, it can be somewhat difficult to discover the problem. Fortunately, *ed* provides commands to eliminate this vagueness. The **h** command explains the last question mark printed by *ed*. The **H** command also explains the last question mark, but also causes a more descriptive explanation of the problem to replace all future question marks. Executing the **H** command again disables the descriptive explanation.

Moving Around in the File

Material Covered:

.	(dot) pointer to the current line;
=	operator; yields line number;
p	command; prints specific lines;
+n	operator; increments dot by <i>n</i> ; default <i>n</i> = 1;
-n	operator; decrements dot by <i>n</i> ; default <i>n</i> = 1;
\$	pointer to the last line of the file;
,	shorthand notation for the range "1,\$";
;	shorthand notation for the range ".,\$";
k	command; creates a pointer to a specific line;
/.../	command; initiates a forward search for the string of characters enclosed between the slashes;
?...?	command; initiates a backward search for the string of characters enclosed between the question marks;
.	metacharacter; matches any single character when used in a search string;
\n	metacharacter; strips away the special meaning (if any) of the character <i>n</i> when used in a search string;
\$	metacharacter; when specified as the last character in a search string, matches the string at the end of a line;
^	metacharacter; when specified as the first character in a search string, matches the string at the beginning of a line;
n*	metacharacter; matches zero or more adjacent occurrences of the character <i>n</i> when used in a search string;
[...]	metacharacters; match any one of the characters enclosed between them when used in a search string;
^	metacharacter; stands for "any character except" when specified as the first character inside [...], causing the braces to match any one character <i>not</i> enclosed between them;
\{...\}	metacharacters; match a specified number of occurrences of the single character enclosed between them when used in a search string.

Your position in a file is always relative to a specific line. *Ed* does not provide commands that move you from character to character. There are five commands that enable you to reference specific lines in a file.

Line Pointers

Of the five commands mentioned above, three are pointers to specific lines in the file.

Pointer to the Current Line

Ed maintains a line pointer called dot (*.*), which always points to the current line in the file. The current line is defined to be the last line affected by an *ed* command. The following table lists some of the more common *ed* operations, and the value of dot after these operations have been performed:

After this operation...	Dot points to...
Invoking <i>ed</i>	Last line of file.
Search for pattern	Closest line containing pattern, relative to your previous position.
Delete last line of file	New last line of file.
Delete line(s) other than last line	Line following last deleted.
Appending, inserting, or changing text	Last line entered.
Read from a file	Last line read in.
Write to a file	Your previous position: dot is not changed.
Substitute new text for old text	Last line affected by substitution.
Execute a shell command	Your previous position: dot is not changed.
Set a line pointer	Your previous position: dot is not changed.
Any unsuccessful or erroneous command	Your previous position: dot is not changed.

Dot can be used as a line number argument for *ed* commands. Assuming you are still editing **test-file**, type

```
.p  
to the file.
```

The **p** command prints specific lines from the *ed* buffer, thus **.p** prints the current line. Note that dot is automatically set to the last line of the file when you first begin editing. You can also specify a range of line numbers with dot. For example,

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

important to note that *ed* always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes to the file.

prints the last four lines of the file. Has the value of dot changed? No, because the last line affected by the **p** command was still the last line of the file. Now try

.-5,-3p

your input to the text located in a special buffer where ed keeps a copy of the file you are editing. It is `\ \ *`. important to note that ed always makes changes to the

which prints the fifth line before dot to the third line before dot. What is dot's value now? Find out by typing

.p

important to note that ed always makes changes to the

Dot is now set to the last line affected by the previous **p** command.

Note that dot need not be typed when specifying ranges. Whenever *ed* sees the + and - operators, *ed* assumes that they refer to the current value of dot. For example,

-2, + 2p

your input to the text located in a special buffer where ed keeps a copy of the file you are editing. It is `\ \ *`. important to note that ed always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes

prints the range of lines from two lines before dot to two lines after dot. Dot is set to the last line printed.

The + and - operators can be used independently to increment or decrement dot by one, respectively. For example, the command

--, +p

important to note that ed always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes to the file.

prints the range of lines from dot decremented by two to dot incremented by one. Also, you can step forward through your text, one line at a time, with a series of plus signs, or step backward with a series of minus signs. Note that [RETURN] is equivalent to +. [RETURN] increments dot by one and prints the resulting current line.

The **p** command provides one other shortcut. Whenever a line number, or one or more operators pointing to a line, appear on a line by themselves, the **p** command is assumed. Some examples are:

```
8
original file are not changed until you write the changes
-----
ed keeps a copy of the file you are editing. It is \ \ *.
+ +
copy of yourrr file in the buffer. The contents of the
```

If a range appears on a line by itself, only the last line of the range is printed. For example,

```
-, +
original file are not changed until you write the changes
```

You can find out the current value of dot by typing

```
. =
8
```

which tells you that dot is currently pointing to the eighth line of the file.

Note that you cannot manually set the value of dot. A command like

```
. = 6
?
```

produces an error. *Ed* reserves to itself the right to change the value of dot, although you may indirectly change dot's value through *ed* commands.

Pointer to the Last Line

Ed also maintains a pointer, called **\$**, which always points to the last line of the file. For example,

```
$
to the file.
```

prints the last line of the file. **\$** can also be used in ranges, as in

```
1,$-6p
```

The *ed* editor operates in two modes: command mode and text entry mode. In command mode, the editor interprets your input as a command. In text entry mode, *ed* adds

which prints the first three lines of **testfile**. Also,

+4,\$p

copy of your file in the buffer. The contents of the original file are not changed until you write the changes to the file.

prints the last three lines of the file. Note that the **+** and **-** operators can apply to **\$** only when **\$** is explicitly typed. By themselves, **+** and **-** always apply to dot.

You can find out the value of **\$** by typing

```
$=  
9
```

which tells you that the ninth line is the last line in the file. Note that **=** does not change the value of dot.

The value of **\$** changes only when a command creates a new last line. **\$** is not user-settable.

Because the "1,\$" and ".,\$" ranges are so commonly used when editing with *ed*, *ed* provides a shorthand notation for each range. The comma can be used in place of "1,\$", so that **,p** prints all the lines in the file. Also, the semicolon means the same thing as ".,\$", so **;p** prints all the lines from the current line to the end of the file.

Setting Pointers to Lines

The **k** command creates a pointer to a specific line, so you can reference that line without knowing its line number. The pointer name must be a lower-case letter. Creating a pointer does not change the value of dot. For example,

```
.  
to the file.  
-4ka  
-2kb  
.  
to the file.
```

creates two pointers, **a** and **b**, which point to the fourth line before dot, and the second line before dot, respectively. Note that dot does not change.

To reference a line with a line pointer you have created, precede its letter name with a single quote ('), as in

`'a,'bp`

`ed` keeps a copy of the file you are editing. It is `\ \ *`.
important to note that `ed` always makes changes to the copy of your file in the buffer. The contents of the

which prints all lines from the line pointed to by `a` to the line pointed to by `b`.

A pointer set by the `k` command always points to the same line, even if that line's line number changes. Thus, the `k` command does not create pointers to specific line numbers, but to specific lines.

Once a pointer has been created, the only way to delete it is to delete the line it points to. Otherwise, that pointer continues to exist until your editing session is over. You can, however, re-assign a pointer to another line, as in

`'ap`

`ed` keeps a copy of the file you are editing. It is `\ \ *`.

`2ka`

`'ap`

text entry mode. In command mode, the editor interprets

which re-assigns `a` to the second line of the file.

You can find out the current line number of a pointer by typing

`'a =`

`2`

`'b =`

`7`

which tells you that `a` is currently pointing to line number 2, and `b` is currently pointing to line number 7.

Searching for Strings

Ed provides a facility which enables you to search for a particular string of characters in your file. A string of characters searched for in this manner is called a *pattern*.

Forward Searches

To initiate a forward search, enclose the pattern between two slashes, and press [RETURN]. For example,

```
/unput/  
your unput as a command. In text entry mode, ed adds
```

searches for the pattern "unput". If the pattern is found, dot is set to the line containing the pattern, and the line is printed on your screen. An unsuccessful search looks like this:

```
/bob/  
?
```

The value of dot is unchanged.

Ed searches forward in your file, starting with the line following the current line. If your pattern has not been found by the time *ed* gets to the end of the file, *ed* wraps around to the beginning of your file and continues looking. *Ed* searches until the pattern is found, or until *ed* reaches the line prior to the starting line of the search.

Backward Searches

You can search backwards in your file by enclosing the pattern between two question marks. For example,

```
?file?  
to the file.
```

searches backwards from the current line, looking for a line containing the string "file". *Ed* found the pattern after wrapping around to the end of the file.

Repeating a Search

Ed remembers the last pattern that was matched. Thus, if you want to repeat a search, you simply type two slashes or question marks. The pattern itself need not be re-typed. For example,

```
?file?  
original file are not changed until you write the changes  
??  
copy of yourrr file in the buffer. The contents of the  
??  
ed keeps a copy of the file you are editing. It is \\ \\*.
```


initiates a backward search for the pattern "file", then finds the next two instances of "file". Note that a repeated search need not be in the same direction as the initial search. For example,

```
/buffer/  
copy of yourr file in the buffer. The contents of the  
??  
your input to the text located in a special buffer where
```

initiates a forward search for "buffer", then repeats the search backwards.

Line Number Arithmetic with Searches

The + and – operators can be used with searches to position yourself at specific lines. For example,

```
/note/ +  
copy of yourr file in the buffer. The contents of the
```

searches forward for a line containing "note", and positions you on the following line. Also,

```
?text?  
your input to the text located in a special buffer where  
??--
```

The ed editor operates in two modes: command mode and

searches backwards for the second line containing "text", and positions you two lines before it.

Note that, although searches have wrap-around capabilities, the + and – operators do not. Thus, an error results if a + or – operator attempts to increment or decrement dot to values greater than \$, or less than one.

The = operator can be used with forward and backward searches to find the line number referred to by the search, as in

```
/unput/=  
3
```

Note that dot is not set to the line containing "unput" in the last example, because = does not change the value of dot.

Using Metacharacters With Searches

There are several characters that have special meaning within the context of a search. These characters, consisting of ., *, [,], ^, \$, \, \{, and \}, are called metacharacters.

The `.` metacharacter matches any single character except a new-line. Thus, the search

```
/.nput/  
your unput as a command. In text entry mode, ed adds  
//  
your input to the text located in a special buffer where
```

first matches "unput" in line 3, and then, when repeated, matches "input" in line 4.

The `*` metacharacter matches zero or more occurrences of the character immediately preceding it. For example,

```
/your*/  
ed keeps a copy of the file you are editing. It is \ \ *.
```

matches "you" in the line displayed. *Ed* stops searching when it finds the first string of characters that matches the given pattern. Thus, "your" or "yourrr" can also be matched with the above search, depending on the current line when the search is initiated.

The last example shows that, even though an "r" is explicitly typed in `/your*/`, there need not be an "r" in the string of characters that are actually matched. This is because zero occurrences of the preceding character is considered a legal match when the asterisk is used. Keeping this in mind, consider the search `/r*/`. Is it useful? No, because zero or more r's can be found on every line in the file. If you want to search for one or more r's, type `/rr*/`.

The `\{` and `\}` metacharacters enable you to control how many occurrences of a particular character are matched. For example, the search `/g\{4\}` finds a string of four g's. The integer between the two metacharacters specifies how many instances of the preceding character are to be matched. Note that this construct matches *exactly* four g's, not four or more. Thus, "yourrr" can be matched by

```
/r\{3\}/  
copy of yourrr file in the buffer. The contents of the
```

If you put a comma after the integer, the `\{ ... \}` construct matches *at least* the specified number of occurrences. For example, `/33.3\{4,\}` matches "33.", followed by at least four 3's. Finally, two integers separated by a comma can be placed in the `\{ ... \}` construct to define an inclusive range which specifies the number of occurrences to match. An example is `/-13\{2,5\}1-`, which matches -1331-, -13331-, -133331-, or -1333331-.

The `[` and `]` metacharacters match any one of the characters enclosed between them. For example, `/h[iau]t/` matches "hit", "hat", or "hut". A range of characters can be specified by typing the beginning and ending character of the range, separated by a minus sign. An example is `/[a-zA-Z][0-9][0-9]*/`, which searches for a single upper- or lower-case letter, followed by one or more digits (the `*` applies only to the `[...]` construct immediately preceding it). The minus sign loses its special meaning within the `[...]` construct if it occurs at the beginning (after an initial `^`, if any), or at the end of the character list.

If the first character after the left bracket is a circumflex (^), then the [...] construct matches any single character *not* included between the brackets. For example, `/[0-9][^0-9]*/` matches one or more occurrences of any character except a digit. The ^ has special meaning in the [...] construct only when it is the first character after the left bracket.

Note that the metacharacters `.`, `*`, `[`, `\`, `$`, `\{`, and `\}` have no special meaning when listed within the [...] construct. Also, the right bracket does not terminate the construct if it is the first character listed after the left bracket (after an initial ^, if any). For example, `/[a-r]/` searches for a single right bracket, or a lower-case letter in the range a through r.

The ^ is also special when typed at the beginning of a string within a search, and requires that the string be matched at the beginning of a line. For example,

```
/^ed/  
ed keeps a copy of the file you are editing. It is \ \ *
```

searches for a line beginning with "ed". The ^ is special only when typed at the beginning of a search string. If ^ is embedded in a pattern, or if it is the only character in the pattern, it is matched literally.

The various ways to use ^ can be illustrated with `/^[^a-z]/`. The first ^ means "match the following pattern at the beginning of a line". The second ^ is literal; it has no special meaning. The third ^, as the first character inside the brackets, means "match any single character except". Thus, this search looks for a ^, followed by any single character except a lower-case letter, occurring at the beginning of a line.

The \$ metacharacter is special when typed at the end of a string within a search, and requires that the string be matched at the end of a line. For example,

```
/and$/  
The ed editor operates in two modes: command mode and
```

searches for a line ending with "and". Also, `/^TEST$/` searches for a line consisting of the single word "TEST".

The \$ is special only when typed at the end of a search string. When embedded in the string, the \$ is matched literally.

The \ (backslash) metacharacter is used to strip away the special meaning associated with a metacharacter. This is useful when you need to match a metacharacter literally in a string. To strip away the special meaning of a metacharacter, simply precede it with \. For example,

```
/\ \ \ \ \ * \ . $/  
ed keeps a copy of the file you are editing. It is \ \ *
```

matches the string "\ *." at the end of a line. Note that \ itself must also be preceded with \ to be matched literally. If you attempt to match the string without using the \ (as in /\ *.\$/), *ed* interprets the search to mean "search for zero or more occurrences of a backslash followed by any single character at the end of a line", which is obviously not what you want. Also,

```
/file\.$/
to the file.
```

matches "file." at the end of a line. If you are ever in doubt about whether or not a character has special meaning, it is safe to precede it with \ just to be sure. If the character has no special meaning, then the \ is ignored.

Adding, Deleting, and Correcting Text

Material Covered:

l	command; list specific lines;
n	command; print lines with line numbers;
a	command; append lines of text after current line;
i	command; insert lines of text before current line;
d	command; delete lines of text;
c	command; change lines of text;
m	command; move lines of text;
t	command; copy lines of text;
j	command; join lines together;
s	command; substitute new text for old text;
g	command; global; perform command list on selected lines of entire file;
G	command; interactive global: on each line selected in the entire file, perform a user-specified command;
v	command; global; perform command list on all lines <i>not</i> selected in the entire file;
V	command; interactive global: on each line <i>not</i> selected in the entire file, perform one user-specified command;
u	command; reverse the most recent modification to the buffer;
\ (... \)	metacharacters: used in left-hand side of s command to break up pattern into pieces that can be referenced individually;
%	metacharacter: used in right-hand side of s command to duplicate right-hand side of most recent s command;
&	metacharacter: used in right-hand side of s command to duplicate left-hand side of same s command.

Printing Lines

Besides **p**, there are two other commands that enable you to print specific lines in the *ed* buffer. The **l** (list) command is similar to **p**, but gives you slightly more information. The **l** command enables you to see characters that are normally invisible. Backspace and tab are represented by overstrikes, and other invisible characters, such as bell, vertical tab, and formfeed, are represented by `\nnn`, where *nnn* is the octal equivalent of the character in the ASCII character set.

The **l** command also breaks long lines into smaller lines of 72 characters each. Thus, if you have lines of text in a file that are longer than 72 characters, **l** breaks them down into 72-character lines so they can fit on your screen. A `\` is printed at the end of each line that is broken.

Print out the contents of **testfile** with the **l** command, and look for any invisible characters:

```
,l
```

```
The ed editor operates in two modes: command mode and
text entry mode. In command mode, the edytor interprets
your unput as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is \ \ *.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
to the file.
```

If you did not make any typing errors that could produce invisible characters, the output looks as shown above. Note that a carriage return and a line feed are not considered invisible, since the placement of text on your screen indicates their presence.

Since some invisible characters can cause strange terminal behavior, you almost always want to eliminate them from your text. This is where the **l** command can save you time and effort by making these characters visible.

The **n** (number) command also enables you to print specific lines, but differs from **p** and **l** in that each line is preceded by its line number and a tab character. Try printing out the contents of **testfile** with **n**:

```
,n
```

```
1 The ed editor operates in two modes: command mode and
2 text entry mode. In command mode, the edytor interprets
3 your unput as a command. In text entry mode, ed adds
4 your input to the text located in a special buffer where
5 ed keeps a copy of the file you are editing. It is \ \ *.
6 important to note that ed always makes changes to the
7 copy of yourrr file in the buffer. The contents of the
8 original file are not changed until you write the changes
9 to the file.
```

Note that the line numbers and tab characters are display enhancements only, and do not become part of the text in the *ed* buffer.

The **p** command is the most common command used to print lines in the *ed* buffer. Keep in mind, however, that wherever it is legal to use the **p** command, the **l** and **n** commands may also be used. The **l** and **n** commands leave dot pointing to the last line printed.

Appending Text

The **a** (append) command appends one or more lines of text after the specified line. By default, the lines of text are added after line dot. Dot is left pointing to the last line appended. After the **a** command is typed, everything you enter is appended to the specified line. To stop appending text, type a period at the beginning of a line, all by itself. This terminates the **a** command, and returns you to command mode. For example,

```
0a
The ed editor is a simple, easy-to-use text editor.
```

```
.
1,3p
The ed editor is a simple, easy-to-use text editor.
The ed editor operates in two modes: command mode and
text entry mode. In command mode, the edytor interprets
```

The **a** command is one of the few *ed* commands that accepts 0 as a line number, enabling you to add text to the beginning of the file, as above. Note that the period at the beginning of an empty line terminates the appended text. The following example can easily occur by forgetting to type the terminating period (*do not try this example!*):

```
$a
It is always comforting to know that your original
file remains intact until you are sure you want to
change it.
1,$p
$-4,$p
;l
.
$-7,$p
original file are not changed until you write the changes
to the file.
It is always comforting to know that your original
file remains intact until you are sure you want to
change it.
1,$p
$-4,$p
;l
```

This poor user typed in the three lines of text that he wanted to append to the end of his file, and then attempted to print out the results. *Ed*, however, was still appending text, and calmly added the user's commands to the file. The user finally realized his mistake, typed the solitary period, and printed out the last eight lines of his file, three of which were the three commands he attempted to execute. The moral of the story is: **REMEMBER THE PERIOD!**

If you type the **a** command and then change your mind, simply type a solitary period on the next line. This terminates the **a** command and adds no lines to the file. Dot is left pointing to the line you specified when you typed the **a** command.

Inserting Text

The **i** (insert) command is similar to the **a** command, except that the added text is inserted before the specified line. By default, the added text is inserted before line dot. Dot is left pointing to the last line inserted. Like the **a** command, the inserted text is terminated by a solitary period at the beginning of a line. For example,

```
2i
Also, it takes very little time to learn.
```

```
.
1,3p
The ed editor is a simple, easy-to-use text editor.
Also, it takes very little time to learn.
The ed editor operates in two modes: command mode and
```

If you type the **i** command and then change your mind, simply type a solitary period on the next line. This terminates the **i** command and adds no lines to the file. Dot is left pointing to the line you specified when you typed the **i** command.

Deleting Text

The **d** (delete) command deletes one or more lines of text from the file. If no lines are specified, line dot is deleted. After a deletion, dot is left pointing to the line following the last line deleted. If the last line of the file is deleted, dot points to the new last line. For example,

```
$d
a
on top of the original contents of your file.
.
$-1,$p
original file are not changed until you write the changes
on top of the original contents of your file.
```

The current last line is deleted, and a new one is typed in its place using the **a** command. The **a** command is used because dot is left pointing at the new last line after the deletion. Thus, it is convenient to append after dot to create the desired last line.

The **d** command can delete several lines at once by specifying a range of lines, as follows:

3,6d

.p

The ed editor is a simple, easy-to-use text editor.

Also, it takes very little time to learn.

ed keeps a copy of the file you are editing. It is `\ \ *`.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
on top of the original contents of your file.



This shows that **testfile** currently contains 7 lines of text, since lines 3 through 6 have been deleted.

Undoing Commands

The **u** (undo) command reverses the effect of the most recent command that made a change to any of the text in the buffer. Use it now to restore the four lines you just deleted:

u

.p

The ed editor is a simple, easy-to-use text editor.

Also, it takes very little time to learn.

The ed editor operates in two modes: command mode and
text entry mode. In command mode, the edytor interprets
your unput as a command. In text entry mode, ed adds
your input to the text located in a special buffer where
ed keeps a copy of the file you are editing. It is `\ \ *`.
important to note that ed always makes changes to the
copy of yourrr file in the buffer. The contents of the
original file are not changed until you write the changes
on top of the original contents of your file.



Note that the **u** command reverses only the most recent command that modified text. Commands that have been succeeded with one or more other commands cannot be reversed with **u**. Besides **d**, **u** also reverses the **a**, **i**, **c**, **g**, **G**, **v**, **V**, **j**, **m**, **r**, **s**, and **t** commands. Dot is left pointing to the last line affected by the reversal.



Changing Lines

The **c** (change) command replaces one or more lines with the text you specify. The **c** command is a combination of the **d** and **i** commands, in that the specified lines are deleted, and the text you type in is inserted in their place. Like the **a** and **i** commands, the replacement text is terminated with a solitary period at the beginning of a line. Dot is left pointing to the last line of replacement text typed in. For example,

1,2c

The ed editor is easy to learn and easy to use.

.

1,3p

The ed editor is easy to learn and easy to use.

The ed editor operates in two modes: command mode and text entry mode. In command mode, the edytor interprets

In this example, the first two lines are deleted and replaced with a single line. Of course, you can also replace a single line with several lines, as in

2c

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The ed editor operates in two modes: command mode and

.

1,/text/p

The ed editor is easy to learn and easy to use.

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The ed editor operates in two modes: command mode and text entry mode. In command mode, the edytor interprets

which replaces the second line of the file with five lines.

If you type the **c** command and then change your mind, simply type a solitary period at the beginning of the next line. This terminates the **c** command with no changes made, and leaves dot pointing to the first line you specified when you typed the **c** command.

Moving Lines

The **m** (move) command moves one or more lines to a new position in the file. By default, **m** moves line dot. Dot is left pointing to the last line moved. For example,

```
2,5m$
```

```
.p
```

The ed editor is easy to learn and easy to use.

The ed editor operates in two modes: command mode and text entry mode. In command mode, the edytor interprets your unput as a command. In text entry mode, ed adds your input to the text located in a special buffer where ed keeps a copy of the file you are editing. It is \ \ *. important to note that ed always makes changes to the copy of yourrr file in the buffer. The contents of the original file are not changed until you write the changes on top of the original contents of your file.

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

which moves lines two through five to the end of the file. Note that **m** appends the moved lines after the specified line. Thus, line number zero is legal as a destination line number, enabling you to move lines to the beginning of the file. The destination line cannot be one of the lines being moved.

Note that the **m** command, as well as any command that accepts line number arguments, accepts pattern searches and line pointers (set by the **k** command) to reference specific lines. For example, **2,/user/++m\$** has the same effect as **2,5m\$** in the previous example. Using pattern searches and line pointers becomes more valuable when you edit large files.

Copying Lines

The **t** command copies one or more lines and places the copy at a specified location in the file. By default, **t** copies line dot. Dot is left pointing to the last line copied, in its new location. For example,

```
1t$
```

```
.-4,$-1t1
```

```
.p
```

The ed editor is easy to learn and easy to use.

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The ed editor operates in two modes: command mode and text entry mode. In command mode, the edytor interprets your unput as a command. In text entry mode, ed adds

your input to the text located in a special buffer where ed keeps a copy of the file you are editing. It is `\ \ *`. It is important to note that ed always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes on top of the original contents of your file.

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The ed editor is easy to learn and easy to use.

This example copied the first line and moved it to the end of the file. Then, the four lines before the new last line were copied and moved after the first line of the file, producing the text shown above.

The only difference between the **m** and **t** commands is that **t** copies the indicated lines and moves them to a new position, leaving the original lines intact. The **m** command moves the specified lines from their original position to a new position. No new text is created.

Modifying Text Within a Line

The **s** (substitute) command is the only *ed* command that enables you to change one or more characters within a line, without having to type the line over again. By default, **s** modifies text on line dot. Dot is left pointing to the last line in which a modification has occurred.

The **s** command enables you to correct the mistakes in your file. Of course, you could use the **d** and **i** commands and re-type each line containing an error, but that is more work than is necessary. For example,

/text/

text entry mode. In command mode, the editor interprets

s/text/text/p

text entry mode. In command mode, the editor interprets

All **s** command lines are of the form

s/replace this/with this/

Thus, the above example first searches for the line containing the string "text", and then replaces "text" with "text" on that line. Note that the **p** command is appended to the **s** command to verify that the intended substitution took place.

Note that the pattern search in the previous example can be included on the **s** command line. The **s** command accepts one line number, to perform a specific replacement on a single line, or two line numbers separated by a comma, to perform a replacement on a range of lines. For example,

```
/unput/s//input/p
```

your input as a command. In text entry mode, *ed* adds

which searches for the pattern "unput" and replaces it with "input". Another feature is illustrated in the above example. Note that the *replace this* portion of the **s** command is empty. This is because the *replace this* portion of the **s** command is a pattern search, just like those discussed under *Searching for Patterns*. You recall from that discussion that *ed* remembers the last pattern you searched for. Thus, since "unput" is the last pattern you searched for, it need not be re-typed in the **s** command. *Ed* remembers the pattern and supplies it for you.

Metacharacters can be used in the **s** command. The *replace this* portion recognizes all the metacharacters discussed under *Searching for Patterns*, plus two additional metacharacters, `\(` and `\)`. These two metacharacters are used to break up the *replace this* portion into pieces that can be referenced individually. For example, in line 1 of the file, suppose you want to interchange the phrases "easy to learn" and "easy to use". The obvious way to do that is to re-type the entire line, but there is an easier way:

```
1s/(ea.*rn\) and \(ea.*se\)\/\2 and \1/p
```

The *ed* editor is easy to use and easy to learn.

Although it is hard to read, it is handy to be able to define pieces of patterns and rearrange them in the *with this* portion. In the above example, the entire *replace this* portion matches "easy to learn and easy to use". The first `\(... \)` matches "easy to learn", and the second `\(... \)` matches "easy to use". These pieces are referred to in the *with this* portion with `\n`, where *n* refers to the *n*-th occurrence of a `\(... \)` pair in the *replace this* portion, counting from the left. Thus, the *with this* portion interchanges the two pieces defined in the *replace this* portion.

Here is another example. Suppose you have a file containing information like

```
Alderson, Mike
Anderson, David
Belford, John
Donally, Kyle
```

```
.
.
.
```

and you want to rearrange each name so that the first name is first, followed by the last name. Re-typing each line could take forever, but the task is easy using the `\(` and `\)` metacharacters. The command

```
.s/\([^\,]*\) , \(.*\)\/\2 \1/
```

does the job. The first `\(... \)` pair matches any number of characters except a comma – the last name. The comma-space between each last and first name is explicitly matched. Finally, the second `\(... \)` pair matches any number of any characters – the first name. These pieces are rearranged in the *with this* portion.

Note that the two portions of an `s` command do not have to be delimited by slashes. You can use any character except a space or a new-line, as long as you use the same character throughout the command line. For example, the previous example can be made a bit more clear by using a capital `o` as the delimiter:

```
,sO\([,]*\), \(.*\)O\2\1O
```

You must be careful to choose a delimiter that is not already used in the `s` command line.

The *with this* portion of the `s` command recognizes only the `\` metacharacter, plus two new metacharacters, `&` and `%`. All other metacharacters previously discussed are interpreted literally in this portion.

The `&` metacharacter is recognized only in the *with this* portion, and stands for whatever is matched by the pattern in the *replace this* portion. For example,

```
2s/done/& quickly/p
```

It was designed to enable the user to get his work done quickly

The `&` stands for whatever pattern is matched in the *replace this* portion, so it stands for "done" in this example. Thus, this example replaces "done" with "done quickly". As another example, first add the line "ed is great" to the end of the file:

```
$a  
ed is great  
.
```

Now use `&` to create two sentences out of one:

```
$s/.*/&? &!/p  
ed is great? ed is great!
```

The `&` must be preceded by `\` to be interpreted literally.

The `%` is also recognized only in the *with this* portion, and stands for whatever was specified in the *with this* portion of the last `s` command that was executed. For example,

```
1s/ed editor/ed text editor/p
```

The ed text editor is easy to use and easy to learn.

```
/ed editor/s//%/p
```

The ed text editor operates in two modes: command mode and

```
//s//%/p
```

The ed text editor is easy to learn and easy to use.

In the first **s** command, the *with this* portion has to be explicitly typed out. Thereafter, a **%** is the only character appearing in the *with this* portion, and stands for "ed text editor". Since the replacement text is the same for the remaining **s** commands, it does not need to be re-typed. Note also how **ed**'s pattern memory is utilized, especially in the last **s** command above.

The **%** is special only when it is the only character in the *with this* portion. If **%** is included in a string of one or more characters, it is no longer special. You can also precede the **%** with a **** to cause literal interpretation.

Now that you know all about the **s** command, you can go through and fix the remaining errors in your file. Here are some suggestions:

```
/edy/s//edi/p
text entry mode. In command mode, the editor interprets
+3s/\ \ \ \ \ * \ ./p
ed keeps a copy of the file you are editing. It is
/yourrr/s//your/p
copy of your file in the buffer. The contents of the
```

Note that, in the second **s** command above, the *with this* portion is empty. This is legal, and is often used when you want to replace erroneous text with nothing at all.

Finally, note that the **s** command operates only on the first occurrence of a pattern on a specified line. Thus, if there are two or more patterns on a line that are identical to the pattern specified in the *replace this* portion, only the first occurrence is actually replaced. The **s** command must be re-executed once for each additional pattern that is to be replaced on the same line.

The **s** command must replace text on at least one of the addressed lines, or **ed** prints a question mark.

Making Commands Effective Globally

The **g** (global) command is used to execute one or more commands on several lines. The lines on which the commands are to be executed are usually specified by pattern searches. The form of a **g** command is

```
x.yg/pattern/command list
```

where *x* and *y* are optional line number arguments, *pattern* is the pattern to be searched for, and *command list* is the list of one or more commands to be executed on each line containing *pattern*. If *x* and *y* are missing, "1,\$" is assumed.

The **g** command first marks every line containing the specified pattern. Then, dot is successively set to each marked line, and the list of commands is executed. If only one command is specified, it is placed on the same line as the **g** command. If several commands are specified, the first command is placed on the same line as the **g** command, and all other commands are placed on the following lines. Every line of a multi-line command list is terminated by `\` except the last. Ending a line with `\` in this way quotes the following new-line, and hides it from the **g** command, thus preventing the new-line from terminating the **g** command prematurely. If no commands are specified, the **p** command is assumed. Any command except **g**, **G**, **v**, and **V** can be used in the command list.

The **g** command can be used as a modifier for the **s** command, enabling the **s** command to replace all the occurrences of a particular pattern on a line, instead of just the first. For example,

```
$/ed/The & editor/gp
```

```
The ed editor is great? The ed editor is great!
```

which replaces both instances of "ed" on the last line with "The ed editor". The **g** command is often used with the **s** command in this way to avoid having to repeat the **s** command once for every additional pattern you want to change on a line. Note that, if the **p** command is omitted, the line is not printed after the substitution is done.

The **g** command becomes more powerful when you specify more than one command to be executed. For example, suppose that you want to change every instance of the string "ed" to "ED", and then mark every line on which the substitution occurs by preceding the line with a series of asterisks. This can be done by typing

```
g/ed/s//ED/g\
```

```
i\
```

```
***
```

```
.p
```

```
***
```

```
The ED text EDitor is easy to use and easy to learn.
```

```
***
```

```
It was designED to enable the user to get his work done quickly  
with the least possible amount of interference from the
```

```
***
```

```
EDitor. This is evident in the lack of prompts and the  
curt error messages.
```

```
***
```

```
The ED text EDitor operates in two modes: command mode and
```

```
***
```

```
text entry mode. In command mode, the EDitor interprets
```

```
***
```

```
your input as a command. In text entry mode, ED adds
```

```
***
```

```
your input to the text locatED in a special buffer where
```

```
***
```

ED keeps a copy of the file you are EDiting. It is

important to note that ED always makes changes to the copy of your file in the buffer. The contents of the

original file are not changed until you write the changes on top of the original contents of your file.

It was designed to enable the user to get his work done with the least possible amount of interference from the

EDitor. This is evident in the lack of prompts and the curt error messages.

The ED text EDitor is easy to learn and easy to use.

The ED EDitor is great? The ED EDitor is great!

This example, though not very useful, illustrates how the **g** command can be used to perform a script of *ed* commands on specific lines. Note that the **g** command accepts as input all lines up to and including the first line that does not end in `\`. Thus, the first line that is not part of the **g** command above is the line containing `,p`. Note also that the period that usually must be typed to end the **i** command is not necessary if the line containing the period is also the last line of the **g** command. Thus, the period, along with the line on which it is typed, can be omitted.

A **g** command can be included in a **g** command list only when it is part of another command, as illustrated in the last example. It is illegal to try to nest command lists by specifying **g** command lists within other command lists.

The **v** command is identical to the **g** command, except that the command list is executed on all lines that do *not* contain the specified pattern.

If the results of a **g** command are not exactly what you had in mind, you can use the **u** command to restore your text to its previous state.

u

.p

The *ed* text editor is easy to use and easy to learn.

It was designed to enable the user to get his work done quickly with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The *ed* text editor operates in two modes: command mode and text entry mode. In command mode, the editor interprets your input as a command. In text entry mode, *ed* adds your input to the text located in a special buffer where *ed* keeps a copy of the file you are editing. It is

important to note that `ed` always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes on top of the original contents of your file.

It was designed to enable the user to get his work done with the least possible amount of interference from the editor. This is evident in the lack of prompts and the curt error messages.

The `ed` text editor is easy to learn and easy to use.

The `ed` editor is great? The `ed` editor is great!

Note that the `u` command also reverses itself, so you can follow one `u` command with another to get back text that you have already reversed.

The `G` (interactive global) command is used when you have one command to execute on each line containing a specific pattern, but this command varies depending on the line. The `g` or `v` command is not appropriate in this case, since the command list for these commands is constant.

The `G` command is invoked in the form

```
x,yG/pattern/
```

where `x` and `y` are line number arguments (if not specified, "1,\$" is assumed), and `pattern` is the particular pattern you want to match in a line. `G` first marks every line containing a string that matches `pattern`. Then, `.` is successively set to each marked line, and the resulting current line is printed on your screen. After the current line is printed, `G` waits for you to enter any single command, and the command you enter is executed. You may specify any command except the `a`, `i`, `c`, `g`, `G`, `v`, or `V` commands. Note that your command can address and affect lines other than the current line. A new-line is interpreted to be a null command. The `G` command can be terminated prematurely by pressing `[DEL]` or `[BREAK]`; otherwise it terminates normally when all lines in the file have been scanned for a string matching `pattern`.

Here is an example:

```
G/editor/
```

The `ed` text editor is easy to use and easy to learn.

```
s/easy/simple/
```

editor. This is evident in the lack of prompts and the

The `ed` text editor operates in two modes: command mode and

```
s/The ed text editor/ed/
```

text entry mode. In command mode, the editor interprets

```
s/the editor/ed/
```

editor. This is evident in the lack of prompts and the

The `ed` text editor is easy to learn and easy to use.

```
s/easy to use/simple to use/
```

The `ed` editor is great? The `ed` editor is great!

```
s[~?]*? //
```

In this example, **G** looks for all the lines containing "editor", and executes the commands you specify. Note that a new-line was typed on each of the two blank lines above, causing no command to be executed.

The **&** character can be typed in place of a command. This causes the most recent command executed within the current invocation of **G** to be re-executed.

The **V** command is identical to the **G** command, except that the lines that are marked and printed are those that do *not* contain a string that matches *pattern*.

The **u** command can be used to reverse all the effects of a **G** command.

Joining Lines Together

The **j** (join) command joins two or more lines together. By default, **j** appends line dot+1 to line dot, but you can specify a range of lines to be joined. Note that **j** does not add any white space between the joined lines. Dot is left pointing to the line created after the specified lines have been joined.

As an example, try joining the last two lines of the file together. First, however, you need to shorten line \$-1 so the joined line fits on one line of the screen. Do this by typing

```
$-1s/easy to learn and //p
The ed text editor is simple to use.
```

Now join the last two lines together with

```
jp
The ed text editor is simple to use.The ed editor is great!
s/\./T/ T/p
The ed text editor is simple to use. The ed editor is great!
```

The last **s** command in this example is used to insert two spaces between the two joined lines. Note that the **p** command can be appended to the **j** command to verify that the two lines have been joined.

Splitting Lines Apart

The **s** command can be used to split a single line into two separate lines. This is done by inserting a new-line between the characters where the split is desired. To do this, the new-line must be preceded by **** to avoid terminating the **s** command prematurely. Thus, you can split the two lines that were joined in the previous example into two separate lines with the **s** command (you cannot use the **u** command to split the last line into two lines now – why?). Do this by typing the following:

```
s/\. T/T/p
```

The ed text editor is simple to use. The ed editor is great!

```
s/\.T/.\
```

```
T/
```

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

The ed text editor is simple to use.

The ed editor is great!

The first **s** command gets rid of the extra white space in the sentence (note that the **u** command could have been used here). The second **s** command inserts a new-line between the period and the capital T, thus creating two separate lines. Note that, although the second **s** command takes up two lines, it is actually one command.

Special Ed Commands

Material Covered:

- f** command; set/print currently remembered file name;
- ;** delimiter; set dot's value;
- w** command; writer characters in buffer to file, or read standard output from a shell command;
- r** command; read contents of file into buffer, or read standard output from shell command;
- e, E** commands; begin editing another file, or read standard output from shell command;
- option; silences character counts generated by **w**, **r**, **e**, **E**, or an invocation of *ed*;
- X** command; initiates text encryption mode;
- x** option; initiates text encryption mode.

Finding the Currently Remembered File Name

If you invoke *ed* with a file name argument, *ed* remembers that file name until your editing session is over, or until the file name is changed as a result of commands that are discussed later in this section. The **f** (file name) command enables you to find out at any time what file name *ed* is remembering. For example,

```
f
testfile
```

which tells you that *ed* is remembering **testfile** as the current file name.

The **f** command also enables you to change the current file name. For example, to change the current file name to **file2**, type

```
f file2
file2
```

Ed echoes "file2" so you can verify that the current file is set correctly. Now change the file name back to the current file, or errors could result in later operations:

```
f testfile
testfile
```

If no file name is specified when *ed* is invoked, then *ed* initially remembers no current file name. Thus, this file name must be supplied when using the **w**, **r**, **e**, or **E** commands (discussed later), or it can be set with the **f** command.

Writing Buffer Text Onto a File

The **w** (write) command writes the text contained in the *ed* buffer onto the specified file, or onto the currently remembered file if no file name is specified. If the write is successful, a count of the number of characters written is printed. Dot is left unchanged.

The **w** command accepts zero, one, or two line number arguments specifying the line or lines to be written. If no line number arguments are given, "1,\$" is assumed.

Try the **w** command by typing

```
w
986
```

The previous contents of **testfile** have now been overwritten by the contents of the *ed* buffer. The number 986 tells you that the write was successful, and that 986 characters were written.

Note that the *ed* buffer is not affected by the **w** command. Its contents are still the same. In fact, all of the line pointers (dot, **\$**, and any that you have set) are still pointing to the same lines as they were prior to the **w** command. Thus, you may write out the contents of the *ed* buffer several times during an edit session without disturbing the current state of the editor. It is a good idea to write often, especially if you have been editing a long time and have made many changes. Depending on how often you write, you can be sure that a current version of your file resides in the relative safety of the file system, should a system crash or a power failure eat up whatever data is in the *ed* buffer.

You can tell *ed* to write to a file other than the currently remembered file by typing

```
/^ed;/^on/w file1
561
```

This command writes the range of lines from the line beginning with "ed" to the line beginning with "on" onto the file **file1**. If **file1** exists, its previous contents are completely overwritten by the specified lines of text. If **file1** does not exist, it is created with a file mode of 666 (modified by the current value of the file creation mask, *umask*) and the specified text is written on it. Again, the number returned indicates that *ed* was successful in writing 561 characters on the file.

The semicolon that appears in the last example is new. If a comma had been used to separate the two searches, *ed* would have started the search for a line beginning with "ed" from the current line. After finding that line, however, *ed* would return to the current line to search for the line beginning with "on". The value of dot would be reset only after finding the line beginning with "on", with the result that a single line address is passed to the **w** command, causing a single line to be written. The semicolon causes the value of dot to be set to the line beginning with "ed", so that the second search is carried out with respect to this line, instead of the previous current line. Thus, two addresses are processed, and the correct lines are written. The semicolon can always be used in place of a comma to force dot to be set at that point in the construct.

You can also run shell commands with the **w** command. The shell command is introduced with **!**. For example,

```
w !ls
file1
testfile
986
```

runs *ls* and also writes the current contents of the buffer to the currently remembered file. Note that the output from *ls* appears on your screen, but is not added to the actual contents of the buffer (the listing that appears on your screen may be longer than that shown above). After the listing is produced, *ed* writes the contents of your buffer to the currently remembered file, and reports the number of characters written. Note that there is no way to run a shell command and write to a file other than the currently remembered file with the **w** command. Note also that **!** is illegal if the editor was invoked from a restricted shell (see *rsh*(1) in the HP-UX Reference manual).

The currently remembered file name is set to the file name you specify with the **w** command, if the specified file name is the first file name mentioned since *ed* was invoked. Otherwise, the currently remembered file name is not affected. A shell command introduced with **!** is never remembered as the current file name.

Reading Files Into the Buffer

The **r** (read) command reads the contents of a specified file, or the currently remembered file, if no file is specified, into the *ed* buffer after the specified line. If no line is specified, the contents are read in after line **\$**. Dot is set to the last line read in.

To illustrate the **r** command, first create a new file called **readfile**:

```
w
986
e readfile
?readfile
a
Here is some text that is to be read in.
It is used to illustrate the r command.
.
w
81
```

You now have a file in your working directory called **readfile**, containing the text shown above. Now begin editing **testfile** again, and read in the contents of **readfile**:

e testfile

986

Or readfile

81

1,5p

Here is some text that is to be read in.

It is used to illustrate the **r** command.

The **ed** text editor is simple to use and easy to learn.

It was designed to enable the user to get his work done quickly with the least possible amount of interference from the

This example reads the contents of **readfile** into **testfile** after line 0, or at the beginning of the file. **Ed** responds by printing the number of characters that were read in. The first five lines of the buffer are printed to verify that the text is placed correctly.

You can also run shell commands with the **r** command. The shell command is introduced with **!**. For example,

/curt/r !date

29

6,9p

editor. This is evident in the lack of prompts and the curt error messages.

Thu Jul 22 10:59:13 MDT 1982

ed operates in two modes: command mode and

which reads the output from **date** into **testfile** after the line containing the pattern "curt". The lines surrounding the insertion are printed to verify that the read executed correctly. Note that, unlike the **w** command, the output from the command becomes part of the text in the buffer. Also, the number of characters read from the command is printed on your screen, but the actual output appears only in the buffer. Note that the **!** is illegal if the editor was invoked from a restricted shell.

The currently remembered file name is reset to the file name you specify with the **r** command, if the specified file name is the first file name mentioned since **ed** was invoked. Otherwise, the currently remembered file name is not affected. A shell command introduced by **!** is never remembered as the current file name.

An **r** command can be reversed with the **u** command. Try this now:

u

6,8p

editor. This is evident in the lack of prompts and the curt error messages.

ed operates in two modes: command mode and

Note that the date and time are no longer present in the buffer.

Editing Other Files

The **e** (edit) command discards the entire contents of the *ed* buffer and reads in the specified file. If no file is specified, then the currently remembered file is read. Dot is set to the last line of the buffer.

If you have made any changes to the buffer since the last **w** command, *ed* requires that you precede the **e** command with a **w** command to save the contents of the buffer. If you are sure that you want to discard the contents of the buffer, you can invoke the **e** command a second time. This forces *ed* to discard the buffer contents and read in the new file. For example,

```
e file1
?  
e file1
561
```

The question mark after the first invocation of **e** is to warn you that you have made changes to the current contents of the buffer, and that these changes will be lost if you do not write them on **testfile**. The second invocation of **e** tells *ed* "I don't care! Do it anyway!". *Ed* complies by discarding the current buffer and reading in the contents of **file1**. *Ed* reports to you the number of characters read.

If you are sure that you want to discard the current contents of the buffer without saving them, you can use the **E** (Edit) command. **E** is similar to **e**, except that *ed* does not check to see if any changes have been made to the current buffer. Thus, you do not have to type the **e** command twice.

If you have made several changes to the buffer, and then decide that you do not like what you have done, you can start editing the same file all over again by typing **e** or **E** with no specified file name. This causes the contents of the currently remembered file to be read into the buffer, destroying the previous contents. Of course, if you have written some of the changes you have made to the current file already, there is no quick and easy way to reverse them.

If you specify a file name with the **e** or **E** command, that file name becomes the new current file, and is remembered for future use with **w**, **r**, **e**, or **E**.

You can also execute shell commands with the **e** or **E** command. The shell command is introduced with **!**. For example,

```
E !ls
23
.p
file1
readfile
testfile
```

This example runs the shell command *ls*, and places its output in the *ed* buffer, destroying whatever was in the buffer previously. The number of characters placed in the buffer is printed for you. The actual list of files and the number of characters read into the buffer may be different than those shown above. Note that **!** is illegal if the editor was invoked from a restricted shell. A shell command is never remembered as the current file name.

Silencing the Character Counts

If the character counts that *ed* produces (when *ed* is invoked, or with the **w**, **r**, **e**, or **E** commands) are annoying or are not helpful, they can be silenced with the **-** option. It is specified when *ed* is invoked, as in

```
$ ed - filename
```

The **-** option also suppresses the question mark generated by the **e** and **q** commands whenever they are not preceded by a **w** command (the **q** command is discussed in the next section).

Encrypting and Decrypting Text

Ed provides a feature that enables you to encrypt and decrypt the text in a file so that other users are not able to read your files. The text is encrypted and decrypted by means of the DES encryption algorithm (see *crypt(1)* in the HP-UX Reference manual). To encrypt your text, you must supply a *key*, which is simply a string of one or more characters. The key determines the manner in which the DES algorithm encrypts your text. You *must* remember this key.

The **X** (encrypt) command enables you to encrypt the text in the *ed* buffer. The **X** command accepts no arguments, but prompts you to enter a key. The echoing on your screen is disabled while you enter the key, so there is no visible record of it. For example,

```
E file1
```

```
561
```

```
.p
```

```
editor. This is evident in the lack of prompts and the  
curt error messages.
```

```
The ed text editor operates in two modes: command mode and  
text entry mode. In command mode, ed interprets  
your input as a command. In text entry mode, ed adds  
your input to the text located in a special buffer where  
ed keeps a copy of the file you are editing. It is  
important to note that ed always makes changes to the  
copy of your file in the buffer. The contents of the  
original file are not changed until you write the changes  
on top of the original contents of your file.
```

```
X
```

```
Enter file encryption key:
```

```
w
```

```
561
```

```
q
```

```
$
```


This example edits **file1**, and prints out its contents. After the **X** command is invoked, you are prompted to enter a key. This key can be any string of characters, but whatever it is, *do not forget your key!* When the **w** command is invoked, the text in the buffer is encrypted according to the key you entered and written on **file1**. The **q** command, which is discussed later, exits the editor and leaves you at the shell level. Now execute the *cat* command to try to print out the contents of **file1**:

```
S cat file1
    (garbage)
    .
    .
    .
$
```

You probably got a screenful of garbage. If your bell beeped a couple of times, this is because the text is encrypted into invisible characters as well as visible characters. There is no practical way for another user to tell what is actually contained in your file.

To edit a file containing encrypted text, use the **-x** option when *ed* is invoked:

```
$ ed -x file1
Enter file encryption key:
561
.p
```

editor. This is evident in the lack of prompts and the curt error messages.

The *ed* text editor operates in two modes: command mode and text entry mode. In command mode, *ed* interprets your input as a command. In text entry mode, *ed* adds your input to the text located in a special buffer where *ed* keeps a copy of the file you are editing. It is important to note that *ed* always makes changes to the copy of your file in the buffer. The contents of the original file are not changed until you write the changes on top of the original contents of your file.

The **-x** option is the same as the **X** command, except that it is used when you invoke *ed*. When prompted for the key, you must enter the same key that you entered when the text was encrypted. Otherwise, the text in that file is inaccessible. This is why it is so important that you remember your key. After the key is entered, the text in **file1** is decrypted and read into the *ed* buffer. You may now edit the text normally.

When you are done editing, if you invoke the **w** command to write your changes to the file, the text is encrypted according to your key. If you want to change your key or disable encryption altogether, you must use the **X** command. When you are prompted for your key, either type in your new key to change the encryption key, or simply type a new-line. If you type a new-line, a null key is entered, and encryption is disabled. Disable encryption now by typing

```
X
Enter file encryption key: (new-line)
w
561
```

The contents of **file1** are now in a readable form.

Note that, when encryption is enabled, all subsequent **e**, **r**, and **w** commands encrypt the text in the *ed* buffer.

As a general rule, text encryption is seldom needed by the typical user except when extreme security is required. The HP-UX file system has its own security system which is sufficient for most security needs. Using text encryption often and/or on several files at once is a dangerous practice, since you must remember your key to successfully edit these files. You should therefore exercise caution when using the text encryption feature.



The Shell Interface

Material Covered:

- ! command: execute shell command;
- q command: exit editor after checking for changes to the buffer;
- Q command: exit editor without checking buffer for changes.

Escaping to the Shell Temporarily

The **!** command enables you to execute a shell command from within the *ed* editor. To do this, type a **!**, followed by the shell command. For example,

```
!(date;who) >whofile
!
```

executes the *date* and *who* commands, and redirects their output into the file **whofile**. Note that *ed* returns a **!** to tell you when the command has completed execution.

If the character **%** appears anywhere in the shell command, it is replaced with the currently remembered file name. Thus,

```
!sort % >sortedfile
sort file1 >sortedfile
!
```

sorts (in reverse alphabetical order) the current contents of **file1**. Note that the current contents of **file1**, not the *ed* buffer, are sorted. The sorted version of **file1** is redirected to the file **sortedfile**. The I/O redirection in the last two examples is used so that the output from these shell commands does not clutter up your screen while you are editing. Note that, if the output from a shell command is printed on your screen, the output does not become part of the *ed* buffer unless **!** is used with the **r**, **e**, or **E** commands.

A final feature of the **!** command is the ability to re-execute the last shell command you executed with **!**, without having to re-type the entire command. This is done by typing two exclamation points, as in

```
!!
!
```

which re-executes the last shell command executed within the *ed* editor. Thus, **sort % >sortedfile** is re-executed.

If a shell command contains any metacharacters, *ed* echoes the command line back to you with all metacharacters expanded (this is what *ed* did in the first *sort* example above). For example,

```
!cat * >bigfile
cat file1 readfile sortedfile testfile whofile >bigfile
!
```

which echoes the expanded command line, then executes the command.

Exiting the Editor

The **q** (quit) command exits the editor. The contents of the buffer are not automatically written on the current file. If you have made any changes to the buffer since the last time you invoked the **w** command, *ed* requires that you issue the **w** command before exiting with **q**. Invoking the **q** command a second time forces *ed* to let you exit without writing the contents of the buffer on the current file. To illustrate this command, first add some text to the buffer, then try to exit without writing:

```
$a
Here is some extra text.
.
q
?
q
$
```

A change is made to the buffer by adding a single line of text to the end of the buffer. When the first **q** command is typed, *ed* sees that there have been changes to the buffer since the last write, so *ed* issues a question mark. This warns you that there are changes to the text in the buffer that will not be saved if you exit without writing. The second **q** command forces *ed* to discard the contents of the buffer and exit. Be very sure that this is what you want to do, since you cannot recover the buffer contents once you have exited. The **\$** is the default shell prompt, indicating that you are once more at the shell level (your shell prompt may be different).

If you know that you want to discard the contents of the buffer and exit, but you do not want to type the **q** command twice, use the **Q** command. The **Q** command is similar to **q**, but *ed* does not check to see if changes have been made to the contents of the buffer.

The **-** option previously discussed disables the question mark that *ed* issues when you do not write before executing an **e** or **q** command. You are living dangerously when it is disabled, however. That question mark has kept many users from accidentally throwing away hours of work. Besides, the **E** and **Q** commands are implemented for those special cases when you want to discard the contents of the buffer.

Miscellaneous Topics

Material Covered:

[DEL],[RUB],[BREAK] keys; any of these keys generates an interrupt signal to *ed*;

Editing Scripts

Interrupting the Editor

[DEL], [RUB], or [BREAK] causes *ed* to stop whatever command it is executing and return to you for a command. *Ed* tries to restore the state of your file to whatever it was before the command was issued. This is easily done if *ed* is interrupted while printing, since dot is not set until printing is done. If *ed* is reading or writing files, or performing substitutions or deletions, however, the state of the buffer (and the current file) is unpredictable; dot may or may not be changed. Thus, it is usually safer to let *ed* finish whatever it is doing, rather than risk finding the buffer or the current file in some garbled state.

Editing Scripts

An editing script is simply a file containing a list of *ed* commands. If you have several files on which a specific list of commands must be executed, it is easier to use an editing script than it is to invoke *ed* once for every file, and perform the tasks in each.

Suppose you have several files named **file1**, **file2**, ..., and you want to perform some specific substitutions, additions, and deletions on each. First, create a file (called **script**, for example), and put all the *ed* commands that you want to execute, in the order that they must be executed, in the file:

```
$ ed script
?script
a
0r !date
1s/.*$/& DATE OF LAST UPDATE/
$-3,$d
g/Karl Harrison/s//Georgia Mitchell/
w
q
.
w
87
q
$
```

The file **script** now contains *ed* commands to put the current date and time at the beginning of each file, append "DATE OF LAST UPDATE" to the date and time, delete the last four lines of each file, and replace every instance of "Karl Harrison" in each file with "Georgia Mitchell". Note that the **w** and **q** commands are included so that the script writes the buffer on each file and exits the editor automatically.

To use **script**, invoke *ed* as follows:

```
$ ed - file1 <script
$ ed - file2 <script
  etc.
```

The I/O redirection character **<** causes *ed*, when invoked, to take its input from **script**. Thus, as *ed* is invoked with each file name, that file is edited according to the commands contained in **script**.

Table of Contents

SED: A Non-Interactive Text Editor

Introduction	1
Overview of Operation	2
Command-line Flags	2
Order of Application of Editing Commands	2
Pattern-space	2
Examples	3
Addresses: Selecting lines for editing	4
Line-number Addresses	4
Context Addresses	4
Number of Addresses	5
Examples	5
Functions	6
Whole-line Oriented Functions	6
Example	7
Substitute Function	7
Replacement Special Characters	8
Substitute Function Flags	8
Examples	8
Input-output Functions	9
Examples	10
Multiple-input-line Functions	10
Hold and Get Functions	11
Example	11
Flow-of-Control Functions	12
Miscellaneous Functions	12





Sed

A Non-Interactive Text Editor

Introduction

Sed is a non-interactive context editor that runs on the HP-UX operating system. It is designed to be especially useful in three cases:

- To edit files too large for comfortable interactive editing;
- To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
- To perform multiple “global” editing functions efficiently in one pass through the input.

The remainder of this article explains *sed* operation and use.

Since only a few lines of the input reside in main memory at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and the attendant risk of errors. *Sed* running from a command file is much more efficient than most or all known editors even if that editor can be driven by a pre-written script.

The main interactive editor functions lost when using *sed* are the lack of relative addressing (because of the line-at-a-time operation) and lack of immediate verification that a command has done what was intended.

Sed is a lineal descendant of the HP-UX editor, *ed*. Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed*. even seasoned users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without carefully reading this article.

The most striking family resemblance between the two editors is in the class of patterns (“regular expressions”) they recognize; both are nearly identical; having virtually identical code.

Overview of Operation

Sed copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by command-line flags (discussed later in this section).

The general format of an editing command is:

```
[address1,address2][function][arguments]
```

One or both addresses may be omitted; address format is explained in the Addresses section of this article. Any number of blanks or tabs can separate the addresses from the function. The function must be present; available commands are discussed in the Functions section of this article. Arguments may be required or optional, depending on which function is being used. Refer to Functions section for details.

Tab characters and spaces at the beginning of lines are ignored.

Command-line Flags

Three flags are recognized on the command line:

- **n** tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Functions section)
- **e** tells *sed* to take the next argument as an editing command;
- **f** tells *sed* to take the next argument as a file name. The file should contain editing commands, one to a line.

Order of Application of Editing Commands

Before any input file is opened or any editing is done, all the editing commands are compiled into a form that will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered which is also the general order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command being the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, *t* and *b* (see Functions section). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the *N* command

Examples

The examples shown throughout this tutorial are all based on the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

Example:

The command

```
2q
```

quits after copying the first two lines of input text, and produces the following output:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

Addresses: Selecting lines for editing

Addresses are used to determine which lines in the input text or file(s) are to be affected by editing commands. Addresses can be either line numbers or context addresses.

Several commands can be associated with a single address or address-pair by enclosing the group of commands between a pair of curly braces (“{ }”).

Line-number Addresses

Line-number addresses are used to specify which lines (in numerical sequence) in the input text are to be modified by the associated *sed* editing commands. As each line is read from the input, an integer decimal line-number counter is incremented. This counter is used to match input lines to the addresses in *sed* commands; when the current line number matches the *sed* address, the command is executed on that line. When editing multiple input files, the counter continues to increment with each line, and does not reset as successive files are opened.

As a special case, the character `$` specifies the last line of the last input file.

Context Addresses

A context address is a pattern (regular expression) enclosed between a pair of slashes (`/<regular expression>/`). `<Regular expressions>` recognized by *sed* are constructed as follows:

1. An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
2. A circumflex (^) at the beginning of a regular expression matches a null character occurring at the beginning of a line.
3. A dollar-sign (\$) at the end of a regular expression matches a null character occurring at the end of a line.
4. The characters `\n` match an **embedded** newline character, but not the newline at the end of the pattern space.
5. A period (.) matches any character except the terminal newline of the pattern space.
6. A regular expression followed by an asterisk (*) matches any number (including 0) of adjacent occurrences of the regular expression it follows.
7. A string of characters in square brackets ([]) matches any character in the string, and no others. If, however, the first character of the string is a circumflex (^), the regular expression matches any character **except** the characters in the string and the terminal newline of the pattern space.
8. A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
9. A regular expression between the sequences `\(` and `\)` is identical in effect to the unadorned regular expression, but has side-effects which are described under Substitute Function and in the next item of this list.
10. The expression `\d` means the same string of characters matched by an expression enclosed in `\(` and `\)` earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of `\(` counting from the left. For example, the expression `^\(.*\)\1` matches a line beginning with two repeated occurrences of the same string.

11. The null regular expression standing alone (e.g., `/`) is equivalent to the last regular expression compiled.

To use one of the special characters (`^ $. * [] \ /`) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash “`\`”.

For a context address to “match” the input requires that the whole pattern within the address match some portion of the input line’s pattern space.

Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. Any command having more addresses than the maximum allowed is considered an error.

- If a command has no addresses, it is applied to every line in the input.
- If a command has one address, it is applied to all lines which match that address.
- If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the process is repeated.

Two addresses are separated by a comma.

Examples:

```
/an/           matches lines 1, 3, 4 in our sample text
/an.*ar/       matches line 1
/^an/          matches no lines
/,/            matches all lines
/\./           matches line 5
/r*an/         matches lines 1,3, 4 (number = zero!)
/^(an\).*\1/   matches line 1
```

Functions

All functions are named by a single character. In the following summary, the single-character function name is listed in the first column. Possible arguments are enclosed in angles (< >), followed by an expanded English translation of the single-character name. The second column contains the number of addresses allowed with that function and an expanded English translation of what the function does. The angles around the arguments are **not** part of the argument, and should not be included in actual editing commands.

Whole-line Oriented Functions

d (2 addresses allowed) Deletes all lines matched by address(es) and does not write them to the output. **Side effect:** No further commands are attempted on the corpse of a deleted line. As soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

n (2 addresses allowed) Reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

a \<text> (1 address allowed) Causes <text> to be written to the output after the line matched by its address. This command is inherently multi-line; *a* must appear at the end of each line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character (\) immediately preceding each newline. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

Once an *a* function is successfully executed, <text> gets written to the output regardless of what later commands do to the line which triggered it. The triggering line can be deleted entirely, but <text> will still be written to the output.

<Text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

i \<text> (1 address allowed) Behaves identically to the *a* function, except that <text> is written to the output **before** the matched line. All other comments about the *a* function apply to the *i* function as well.

c \<text> (2 addresses allowed) Deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i*, *c* must be followed by a newline hidden by a backslash; and interior new lines in <text> must be hidden by backslashes.

C can have two addresses, and therefore affect multiple lines. When two addresses are present, all the lines in the range are deleted, but only one copy of <text> is written to the output; **not** one copy per line deleted.

As with *a* and *i*, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a *c* function, no further commands are attempted on the corpse.

If text is appended after a line by a *a* or *r* functions and the line is subsequently changed, the text inserted by the *c* function is placed **before** the text of the *a* or *r* function (*r* is described in the Input-Output Function section).

Note: In output text produced by these functions, leading blanks and tabs are eliminated, as in all *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash (the backslash does not appear in the output).

Example:

The following commands, when applied to our example input text:

```
n
a \
XXXX
d
```

produce:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n                n
i \              c \
XXXX            XXXX
d
```

Substitute Function

The *substitute* function changes parts of lines selected by a context search within the line. Command structure is as follows:

```
s<pattern><replacement><flags>
```

- (2 addresses allowed) The *s* function replaces **part** of a line (selected by <pattern>) with <replacement>. It can best be read:

```
Substitute for <pattern>, <replacement>
```

- The <pattern> argument contains a pattern, exactly like the patterns in <addresses> (discussed previously). The only difference between <pattern> and a context address is that the context address must be delimited by slash (/) characters; <pattern> can be delimited by any character other than space or newline.
- By default, only the first string matched by <pattern> is replaced, but see the *g* flag below.
- The <replacement> argument begins immediately after the second delimiting character of <pattern>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly **three** instances of the delimiting character in a given substitution command.)
- The <replacement> is not a pattern, and the characters which are special in patterns have no significance in <replacement>. Instead, other characters are special:

<Replacement> Special Characters

- `&` is replaced by the string matched by `<pattern>`
- `\<d>` (where `<d>` is a single digit) is replaced by the `d`th substring matched by parts of `<pattern>` enclosed between `\(` and `\)`. If nested substrings occur in `<pattern>`, the `d`th is determined by counting opening delimiters `\(`.
- As in patterns, special characters may be made literal by preceding them with backslash `\`.

Substitute Function Flags

The `<flags>` argument can contain the following flags:

- `g` Substitute `<replacement>` for all (non-overlapping) instances of `<pattern>` in the line. After a successful substitution, the scan for the next instance of `<pattern>` begins just after the end of the inserted characters; characters put into the line from `<replacement>` are not rescanned.
- `P` Print the line if a successful replacement was done. The `P` flag causes the line to be written to the output if and only if a substitution was actually made by the `s` function. Note that if several `s` functions, each followed by a `P` flag, successfully substitute in the same input line, multiple copies of the line will be written to the output; one for each successful substitution.
- `w <filename>` Write the line to a file if a successful replacement was done. The `w` flag causes lines which are actually substituted by the `s` function to be written to a file named by `<filename>`. If `<filename>` exists before `sed` is run, it is overwritten; if not, it is created.
- A single space must separate `w` and `<filename>`.
- The possibilities of multiple, somewhat different copies of one input line being written are the same as for `P`.
- A maximum combined total of 10 different file names can be mentioned after `w` flags and `w` functions.

Examples:

The following command, applied to our standard input,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubhla Khan  
A stately pleasure dome decree:  
Where Alph, the sacred river, ran  
Through caverns measureless by man  
Down by a sunless sea.
```

and, on the file `changes`:

```
Through caverns measureless by man  
Down by a sunless sea.
```


If the nocopy option is in effect, the command:

```
s/[.,;?:]/*P&*/gP
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

Finally, to illustrate the effect of the `g` flag, the command:

```
/X/s/an/AN/P
```

produces (assuming nocopy mode):

```
In XANadu did Kubhla Khan
```

and the command:

```
/X/s/an/AN/gP
```

produces:

```
In XANadu did Kubhla KhAN
```

Input-output Functions

`p`
(print) (2 addresses allowed) Writes the addressed lines to the standard output file at the time the `p` function is encountered, regardless of what succeeding editing commands may do to the lines.

`w <filename>`
(`w r i t e` o n `<filename>`) (2 addresses allowed) Writes the addressed lines to the file named by `<filename>`. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the `w` and `<filename>`.

A maximum combined total of ten different files can be mentioned in write functions and `w` flags after `s` functions.

`r <filename>`
(read the contents of a file) (1 address allowed) Reads the contents of `<filename>` and appends `<filename>` data immediately following the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line that matched the address.

If `r` and `a` functions are executed on the same line, the text produced by the `r` and the `a` functions is written to the output in the order that the functions are executed.

Exactly one space must separate the `r` and `<filename>`. If a file mentioned by an `r` function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

Note

Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in *w* functions or flags; that number is reduced by one if any *r* functions are present. (Only one read file is open at one time.)

Examples

Assume that the file *note1* has the following contents:

```
Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was
the grandson and most eminent successor of Genghiz (Chingiz)
Khan, and founder of the Mongol dynasty in China.
```

The following command:

```
/Kubla/r note1
```

produces:

```
In Xanadu did Kubla Khan
  Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was
  the grandson and most eminent successor of Genghiz (Chingiz)
  Khan, and founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Multiple-input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing embedded newline characters; they are intended principally to provide pattern matches across lines in the input.

- | | |
|---|--|
| N
(next line) | (2 addresses allowed) The next input line is appended to the current line in the pattern space; the two input lines are separated by an embedded newline character. Pattern matches can extend across the embedded newline character(s). |
| D
(Delete first part of the pattern space) | Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline character was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning. |
| P
(Print first part of the pattern space) | Print up to and including the first newline character in the pattern space. |

The *P* and *D* functions are equivalent to their lowercase counterparts if there are no embedded newline characters in the pattern space.

Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

<code>h</code> (hold pattern space)	(2 addresses allowed) Copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).
<code>H</code> (Hold pattern space)	(2 addresses allowed) Appends the contents of the pattern space to the contents of the hold area. The former and new contents are separated by a newline character.
<code>g</code> (get contents of hold area)	(2 addresses allowed) Copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).
<code>G</code> (Get contents of hold area)	(2 addresses allowed) Appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline character.
<code>x</code> (exchange)	(2 addresses allowed) Interchanges the contents of the pattern space and the hold area.

Example

The commands

```
1h
1s/ did,*/
1x
G
s/\en/ :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan :In Xanadu
A stately pleasure dome decree: :In Xanadu
Where Alph, the sacred river, ran :In Xanadu
Through caverns measureless to man :In Xanadu
Down to a sunless sea. :In Xanadu
```

Flow-of-Control Functions

These functions do not alter the input lines, but control the application of functions to the lines selected by the address part.

! (Don't) (2 addresses allowed) Causes the next command (written on the same line), to be applied to all and only those input lines **not** selected by the address part.

{ (Grouping) (2 addresses allowed) The grouping command “{” causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the { or on the next line.

The group of commands is terminated by a matching } standing on a line by itself.

Groups can be nested.

:<label> (place a label) (No address allowed) Marks a place in the list of editing commands that can be referred to by *b* and *t* functions. <Label> can be any sequence of eight or fewer characters. If two different colon functions have identical labels, a compile-time error diagnostic is generated, and no execution is attempted.

t<label> (branch to label) (2 addresses allowed) Causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile-time error diagnostic is produced, and no execution is attempted.

A *b* function with no <label> is taken to be a branch to the end of the list of editing commands. Whatever should be done with the current input line is then done, and another input line is read. The list of editing commands is restarted from the beginning on the new line.

t<label> (test substitutions) (2 addresses allowed) Tests whether **any** successful substitutions have been made on the current input line. If so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

- 1) reading a new input line, or
- 2) executing a *t* function.

Miscellaneous Functions

= (equals) (1 address allowed) Writes to the standard output the line number of the line matched by its address.

q (quit) (1 address allowed) Writes the current line-to-be-written to the output (if it should be), plus any appended or read text to be written, then terminates execution.

Table of Contents

AWK: A Programming Language for Manipulating Data

Introduction	1
The Command Line	2
Structure of Awk Programs	3
Predefined Variables	3
Output	4
Redirecting Output to Files	4
Formatting Output	5
Details of Awk Programming	6
Designing Patterns	6
Regular Expressions and Special Characters	6
Relational Expressions	8
Combinations of Patterns	9
Pattern Ranges	9
Designing Actions	9
Variables	9
Field Variables	10
Arrays	11
Built-in Functions	11
Flow-of-Control Statements	12
Commenting	13
Error Messages	14
Notes on the Design	15
Notes on Awk Implementation	15
Annotated Examples	16
Generating Reports	16
Doing Calculations	19
Rearranging Data	20
References	22





Awk: A Programming Language for Manipulating Data

Introduction

Awk is a useful tool for manipulating data and text. Unlike the HP-UX commands that do similar work, *awk* comprises its own programming language. This lets you process input in various ways, such as:

- Generate reports on the contents of files
- Transform the text or data within files
- Manipulate columnar data
- Search files for specific patterns

With *awk*'s ability to search files and generate reports, you can treat some of your ordinary files as databases. The terminology used in *awk* – “records” and “fields” – reinforces this idea.

The *awk* programming language includes such constructs as **for**, **while**, and **if-else**, as well as a set of built-in functions and variables. The language resembles the C programming language. If you are familiar with C, you should be able to master *awk* almost immediately. If you don't know C, you should still find *awk* easy to learn and use.

Awk is named for its designers: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan, from Bell Laboratories. For a detailed discussion written by these people, read “Awk—A Pattern Scanning and Processing Language,” published by Bell Labs in 1978 and available in many technical libraries.

This article is for the user who is familiar with HP-UX and who has used a programming language. There are examples throughout the article; you should try them as you encounter them. You should take time now to create a small input file, using any of the HP-UX editors, or by typing the following command line:

```
$ cat >hello.awk
```

The *cat* command followed by *>* allows you to type text directly into the file *hello.awk* from the keyboard. The filename is arbitrary; the *.awk* suffix is just a reminder for you and is optional.

Now type in

```
Hello, world!  
Howdy, partner!
```

End the file by typing **CONTROL** and **D** together (the end-of-file character) and then pressing **RETURN**. (**CONTROL** may be marked **CNTL** or **CTRL** on your keyboard; **RETURN** is marked **ENTER** on some keyboards with HP-UX overlays.) The shell prompt should reappear; your example file is ready to use.

The Command Line

You can program *awk* entirely on the same line with the prompt. This is often done in practice. The format is:

```
$ awk 'awk_program' input_filename
```

The dollar sign at the beginning of this command line represents the shell prompt in this tutorial. Your shell prompt may be different.

Command-line *awk* programs must be surrounded by single or double quotes, so that the shell will see the whole program as a single argument to the command. Single quotes prevent the shell from interpreting any special characters you may have included in the *awk* program. All examples in this tutorial use single quotes.

If your *awk* program exceeds one line, you can type a backslash (`\`), then press the **RETURN** key, and continue typing the program. For example:

```
$ awk 'awk_pro\ RETURN  
> gram' in\ RETURN  
> ut_filename RETURN  
output of program  
$
```

The `>` is an auxiliary prompt (which may be different on different systems or shells) that tells you you're still typing a single logical command line.

This maneuver is called *escaping the newline character*. You can use it when invoking any command from the shell.

For some applications you may want to write *awk* programs that are many lines long. It makes sense to store such long programs, and any programs that you often use, in separate files. Note that no compilation step is necessary. The file doesn't have to be executable, just readable.

To invoke *awk* using a program stored in a separate file, use the `-f` option:

```
$ awk -f awk_program_name input_filename
```

You can give an *awk* program input from your keyboard (standard input) by typing a dash ("`-`") instead of an input filename. Keyboard input is terminated by typing **CONTROL-D**. Your command line would look like this:

```
$ awk 'awk_program' -
```

or

```
$ awk -f awk_program_name -
```

An example of this procedure is shown in the Annotated Example, "Doing Calculations."

Structure of Awk Programs

Awk programs are built of one or more *statements* that have the general form:

```
pattern {action}
```

For every line in the input that matches the pattern, the specified action is executed. The action part is always enclosed in braces.

You can specify multiple actions within the action part by separating them with semicolons or newline characters (typing **RETURN** creates a newline character).

Awk processes input one line at a time. For *each* line of input, *awk* scans *all* the patterns in the program. Whenever it finds a pattern that matches the line of input in question, it executes the associated action.

An *awk* statement may consist of the pattern or the action or both. A pattern without an action prints out each input line that matches the pattern (this is the default action); an action without a pattern executes the action on every line of input (the default pattern matches anything).

Predefined Variables

The input is made up of a series of *records*. The default record separator is a newline character: by default, each input line is a record.

Records are divided into *fields*; the default field separator is white space (tabs or blanks). So the input

```
Hello, world!
```

consists of one record (one line) and two fields (the strings “Hello,” and “world!”, which are separated by a blank).

The output is also made up of fields and records. The default output field separator is a blank and the default output record separator is the newline character.

The variables *FS* and *RS* contain the current input field and record separators; the output separators are in *OFS* and *ORS*. You can change any of them at any time by simply assigning them any single character value.

On the command line, you can use the argument **-Fc**, which sets *FS* to the character value *c*. Use assignment statements (such as *RS* = “@”) to specify new values for any of the other predefined variables or as an alternate way to change *FS*. (Be sure to put double quotes around new separators to ensure that they are interpreted correctly.)

Each field is designated by a *field variable*. In the first record of *hello.awk*, the string “Hello,” is stored in the field variable *\$1* and “world!” is stored in the field variable *\$2*. In general, field *n* of the current record is stored in the variable *\$n*. The whole current record is stored in *\$0*.

A predefined variable called *NF* contains the number of fields in the current record. The number of the record currently being processed is stored in *NR*; you can find out how many records are in the input by printing *NR* at the end of your program.

Output

The simplest type of `awk` program prints out each line in an input file that matches a specified string. Try this command:

```
$ awk '/Hello/' hello.awk
```

There is no action supplied here, so each record (line) that contains “Hello” somewhere within it is printed. Note that the string is surrounded by slashes, and that the whole `awk` program is surrounded by single quotes. You must always use these slashes around patterns which consist of strings that are regular expressions (described in the section of this article entitled “Regular Expressions and Special Characters”), and you should use single quotes around a command-line program so the shell will see it as one argument and not attempt to interpret any special characters that may be lurking within the program.

The output for the above command is the matching record:

```
Hello, world!
```

The following program contains an action, but it does the same thing as the above actionless program:

```
$ awk '/Hello/ {print $0}' hello.awk
```

This is an example of the **print** action. Since `$0` refers to the entire record, this program prints every record containing “Hello” on the standard output. To print out the second and first fields, in that order, of each record containing “Hello”, type:

```
$ awk '/Hello/ {print $2, $1}' hello.awk
```

and you’ll get:

```
world! Hello,
```

The comma between the field arguments tells `awk` to put an output field separator (a space by default) between the output fields. Without the comma, the fields would be concatenated (run together).

Redirecting Output to Files

You can send the output of the **print** action into files by using `>` or `>>`. The program

```
$ awk '/Hello/ {print $1 >"file1"; print $2 >"file2"}' hello.awk
```

writes the first field, "Hello,", into *file1* and the second, "world!", into *file2* (creating the files if necessary). You must put double quotes around the file names. The program:

```
$ awk '/Hello/ {print $1 >>"file1"}' hello.awk
```

appends the first field to *file1* rather than overwriting it, so now *file1* contains:

```
Hello,  
Hello,
```

The file name to which you divert your output may also be a variable or a field. The action:

```
$ awk '/Hello/ {print NF > $2}' hello.awk
```

uses *\$2* for the filename. You should take care in cases like this one that the value assigned to the variable is a valid file name. If it is not, you will get an error message and the program will abort.

Formatting Output

You can format your output with the **printf** statement. The *awk* **printf** statement is identical to the **printf** library routine used in the C programming language. The statement's structure is

```
printf format, expr, expr, ..
```

The format for the list of expressions is specified in the *format* argument. **Printf** prints the expressions in the specified format. For example,

```
$ awk '{printf "%7s %10.3f\n", $1, NF}' hello.awk
```

prints *\$1* (the first field) as a seven-character string, and *NF* as a floating point number in a ten-digit field width with three digits after the decimal point. Try this and get:

```
Hello,      2.000
```

The newline character is `\n`, which appears at the end of the *format*. You must specify all spaces, separators, and newlines that you want in the output. Note that you don't have to specify a newline when using **print**, because **print** automatically appends the output record separator (by default, a newline) to its output string.

For a full discussion of **printf**, look in McGilton and Morgan's *Introducing the UNIX™ System*, Kernighan and Ritchie's *The C Programming Language*, or the article "Using the C Library Routines" in Volume 2 of *HP-UX Concepts and Tutorials*. (These are listed in a reference section at the end of this tutorial.)

Details of Awk Programming

The full structure of *awk* programs includes optional statements labeled by the special patterns *BEGIN* and *END*:

```
BEGIN  { action }
.      .
.      .
pattern { action }
.      .
.      .
END    { action }
```

The action in the *BEGIN* statement is executed once before any of the input has been read (hence before any patterns are evaluated). The action in the *END* statement is executed once after all the input has been read. These special statements give you opportunities to set parameters before the program begins or to process or tabulate data after *awk* has seen all of the input. For example,

```
BEGIN {OFS = "@"}
END   {print NR}
```

changes the output field separator to “@” before any input is read, and prints out how many records are found in the input after all of it has been read.

Designing Patterns

You have many options for writing *awk* patterns, including:

- Regular expressions
- Relational expressions
- Combinations of expressions
- Boolean expressions
- Ranges of patterns

You have a complete set of operators and special characters with which to build patterns.

Regular Expressions and Special Characters

Patterns can be made from regular expressions. Regular expressions are always enclosed in slashes. A simple pattern is a string enclosed in slashes:

```
/world/
```

If entered as a program (`$ awk '/world/' hello.awk`) this expression would print out all lines in an input containing occurrences of “world”, both as a field alone (a complete word) and as part of a field, such as “worldly” or “world!”

Between the slashes that delimit regular expressions, you can use most of the standard special characters (or metacharacters) that are recognized by the *ed* editor and by the shell. The available special characters for use between slashes in regular expressions are:

- | perform a logical OR of the regular expressions on either side of the vertical bar.
- + match if there are one or more occurrences of the preceding regular expression.
- ? match if there are zero or one occurrence(s) of the preceding regular expression.
- [] match any of the characters inside the brackets.
- [x-x] match any character in the lexical range bounded by the characters on either side of the dash. The range is enclosed by the brackets.
- ^ match only if the matching regular expression is found at the beginning of the line.
- \$ match only if the matching regular expression is found at the end of the line.
- * match any combination of characters, including a null string.
- .
- \ turn off the special meaning of the next character, so the character can represent itself (a maneuver known as escaping the character).
- () group the evaluation of regular expressions.

For example,

```
$ awk '/^main/' c_program.c
```

matches records beginning with "main", and:

```
$ awk '/Albuquerque;Santa Fe/' article_about_NM
```

matches records containing a reference to either Albuquerque or Santa Fe.

To turn off a special character's special meaning, precede it with a backslash in the expression. For example,

```
/\/.*\//
```

matches any string of one or more characters that is enclosed in slashes.

You can abbreviate a sequence of characters in a string. This is called *character class abbreviation*. For example,

```
$ awk '/[Hh]ello/' hello.c
```

matches:

```
Hello, world!
```

but it would also have matched a record containing:

```
Well, world, hello!
```

The sequence `[a-zA-Z0-9]` would match all letters, both upper and lower case, and all digits. To use such ranges you need to understand how your character set is arranged. McGilton and Morgan explain this in their book.

In patterns, you can specify that a field or variable (an expression) matches a regular expression using the tilde character `~` to mean “match” and `!~` to mean “don’t match.” For example, the pattern:

```
$ awk '$1 ~ /[Hh]ello/' hello,awk
```

matches all records that contain either “Hello” or “hello” in the first field. This pattern also matches records containing “Othello” in the first field. To *reject* records with “Hello”, use

```
$ awk '$1 !~ /[Hh]ello/' hello,awk
```

Relational Expressions

In *awk* patterns, you can use the relational operators `<`, `<=`, `=`, `!=`, `>=`, and `>` between expressions. For example, the pattern

```
$ awk '$2 >= $1 + 100' filename
```

selects lines in which the second field is numerically at least 100 greater than the first field.

Relational operations are always *numeric* comparisons (as in the above example) unless *both* operands are strings; in that case a *string* comparison is made. Fields are treated as strings unless there is information to the contrary, so

```
$ awk '$1 > $2' filename
```

automatically performs a string comparison on the first two fields, matching if *\$1* has a larger character value (in ASCII) than *\$2*.

Note that the regular expressions in the last two examples of the Regular Expressions and Special Characters section match the string “Hello,” in the *hello.awk* file as well as “Hello” or “hello”, or even “helloes”. You can eliminate these various matchings by using a string instead of a regular expression:

```
$ awk '$1 == "Hello," || "hello," / hello,awk
```

matches only if *\$1* is either the string “Hello,” or “hello,” and nothing else. This generates the same output as the program

```
$ awk '$1 ~ /^[Hh]ello,$/' hello,awk
```

which specifies that *\$1* starts with “H” or “h” and ends with a comma. In this case it’s shorter to use the regular expression.

If you use regular expressions in your patterns, you can match many strings. But if you use strings in your patterns, you can match only those exact strings in the input. Both tactics are valuable in different situations.

Combinations of Patterns

You can combine several patterns into one using the Boolean operators `||` (or), `&&` (and), `==` (equal to), and `!=` (not equal to). For example, the pattern

```
$ awk '$1 >= "H" && $1 < "I" && NF == 2 && $2 != "world!"' hello.awk
```

matches records that begin with “H” and have two fields but do not have “world!” as the second field. The record “Hello, world!” won’t match, but the record “Howdy, partner!” (or “Houston, Texas” for that matter) will match.

Awk always evaluates the operands of `&&` and `||` from left to right. The evaluation stops as soon as the expression is found to be true or false. You can use parentheses freely to force the order of evaluation or to increase legibility.

Pattern Ranges

The pattern you specify in a pattern-action statement can consist of two patterns separated by a comma. When you specify the pattern in this way, the action is executed on each record from an occurrence of the first pattern through the next occurrence of the second pattern. For example,

```
$ awk '/Hello/,/Partner/' hello.awk
```

prints all records from the first one matching “Hello” through the next one matching “partner”. The statement

```
$ awk 'NR == 10, NR == 30 {print $0}' filename
```

prints records 10 through 30 of some file (try it on one of your files). If you use the above program on a 20-line file, *awk* will print lines 10 through 20 and stop without generating an error.

Designing Actions

Actions consist of one or more statements. A statement can include:

- Arithmetic expressions
- Assignment statements
- Output statements
- Built-in function calls
- Flow-of-control statements

Variables

In *awk* programs, you do not need to write declaration statements for variables. The variables take on numeric (floating point) or string values automatically, according to context. For example,

```
x = 1           gives x a numeric value;
```

```
x = "HP-UX"     gives x a string value;
```

```
x = "3" + "4"   assigns 7 to x as if the equation were  $x = 3 + 4$  (because context demands numeric values in this case).
```

Variables are automatically initialized to the null string (numerical value = 0), so you don't need to initialize variables in a *BEGIN* statement. For example, the sums of the first two fields of all records can be computed by a two-line program:

```
{ s1 = s1 + $1; s2 = s2 + $2 }
END { print s1, s2 }
```

which you can enter and then try by typing

```
# awk -f two_line_program file_with_numbers
```

Awk does all of its arithmetic internally and in floating point. The available operators are:

+, -, *, and /	addition, subtraction, multiplication, and division
%	the mod operator (for example, the pattern <i>NF % 2 == 0</i> prints all lines in the input that have an even number of fields)
++ and --	the increment and decrement operators (like those in C language)
+=, -=, *=, /=, and %=	the C language assignment operators (for example, <i>x += 1</i> is the same as <i>x = x + 1</i>).

Any of these operators may be used in an expression.

Field Variables

You can treat the field variables (*\$1*, *\$2*, etc.) just as any other variable. You can replace fields with other numbers, assign results to a field, or use fields in expressions. For example,

```
# awk '{ $1 = $2 + $3; print $0 }' filename
```

accumulates fields 2 and 3 into field 1 and prints out the record with a new field 1. If you use *hello.awk* for *filename*, *awk* will convert the strings to numbers in response to the context and *\$1* will turn out to be zero. *\$3* is a null string which equals zero.

Fields can be referred to by numerical expressions, such as *\$(i)*, *\$(n + 1)*, or *\$(NF*4 + 3/(NR - 5))*. (If the expression comes out non-integer, *awk* truncates the decimal portion and uses the remaining integer portion as the result.) For example, to refer to the last field when you're unsure how many fields are in the record, use *\$NF*.

Whether a field variable is considered numeric or string depends on context. The matter is not a concern to most *awk* users. You may run into ambiguous cases such as

```
if ($1 == $2)
```

in which *awk* has no criteria for deciding whether to compare strings or numbers. Just as in the relational expressions discussed earlier, *awk* solves the ambiguity by treating fields as strings in such cases.

Arrays

Awk also defines and initializes arrays automatically. To create an array, simply mention it when you need it; *awk* creates the array for you then and there. The subscripts can have numeric values or string values, such as `x["Hello,"]`. The program

```
/Hello/ {x["Hello"]++}  
/world/ {x["world"]++}  
END     {print x["Hello"], x["world"]}
```

counts the occurrences of “Hello” and “world” in the input, stores the counts in elements of the array, and prints the final results. Enter this program in a file, and try it using the `-f` option.

Built-in Functions

You can use a number of built-in functions in your *awk* programs. These include both string and arithmetic operations.

length(x) returns the length of the argument. For example,

```
$ awk '{print length($1), $0}' hello.awk
```

prints the length of the first field then prints out the entire record.

sqrt(x) returns the square root of the argument.

log(x) returns the base *e* logarithm of the argument.

exp(x) returns the exponential of the argument.

int(x) returns the integer part of the argument.



The arguments of functions can be any expression. For all of the above functions, the name of the function alone, with no argument, will cause the function to be performed on the entire record.

substr(s,m,n) returns the substring of *s* that begins at position *m* and is at most *n* characters long. For example,

```
$ awk '/Hello/ {print substr($2,3,5)}' hello.awk
```

will produce

```
rd!
```

which is the substring of `$2` – “world!” – starting with the third letter – “r” – and is no more than five characters long (the length of this substring happens to be four).

split(s,array,sep) splits the string *s* into `array[1],...,array[n]`. (*s* can be a variable.) The number of elements found is returned as *n*. If you don't provide a field separator in the *sep* argument, the current value of *FS* is used by default.

index(s1,s2) returns the position in which the string *s2* occurs in the string *s1*. If *s2* is not a subset of *s1*, **index** returns a 0. For example,

```
$ awk '/world/ {print index($2,"r")}' hello.awk
```

prints out 3, because “r” is the third character in `$2` (“world!”).

printf(*f*,*e1*,*e2*...) places the values of *e1*, *e2*, and so on into the formatted fields specified by *f*. The argument *f* is the format string, which is like the **printf** format string. For example,

```
$ awk '{x = sprintf("%8s %10s", $1, $2);\n > print x}' hello.awk
```

sets *x* to the string produced by formatting strings *\$1* and *\$2* and prints the result. For a complete discussion of output formatting, look in "Using the C Library Routines" in Volume 2 of *HP-UX Concepts and Tutorials* or Kernighan and Ritchie's *The C Programming Language*.

The other built-in functions that you have already seen are:

print introduced in the Output section of this article

printf introduced in the Formatting Output section

Flow-of-Control Statements

You can use many of the same flow-of-control statements available in C (see *The C Programming Language*). *Awk* provides **if-else**, **while**, and **for** statements, and statement grouping with braces just as in C.

if(*cond*) *stmt* the condition in parentheses is evaluated; if it's true, the statement following the **if** is executed. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement.

else *stmt* The optional **else** statement is executed if the **if** condition is false. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement. For example,

```
$ awk '{if($1 != "Hello,") print $0;\n > else print "Argh!"}' hello.awk
```

prints lines that do not start with "Hello," and prints a complaint when it encounters a line that does. (Note the use of the backslash to fit this long program onto a single command line.)

while(*cond*) *stmt* The condition in parentheses is evaluated; as long as it is true, the statements in braces are executed. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement. The **while** condition is tested before each pass. Try this example:

```
$ awk '{while(i<=2) {print $(i); i++ };\n > i=0}' hello.awk
```

for(*cond*) *stmt* While a variable changes from an initial to a final value, the statement(s) in the braces are executed. Multiple statements are enclosed in braces and separated by semicolons or newlines. The braces are optional if there is only one statement. Here is the format of the condition:

```
for (initialize; final; increment){ ... }
```

Also, you can use:

```
for (i in array) statement
```

This construction executes *statement* for each element in the specified array. The elements are not necessarily accessed in order. Changing *i* or accessing any new elements during the statement will introduce side effects.

The conditional expression used in the **if**, **while**, and **for** statements can contain any of the standard relational operators (<, <=, >, >=) as well as the match operators ~ and !~ and the logical operators ||, &&, ==, and !=. Parentheses for grouping are allowed (and encouraged).

Here are the other flow-of-control statements:

break	exits the current while or for construct.
continue	immediately starts the next iteration of the current loop.
next	causes <i>awk</i> to skip immediately to the next record and begin scanning the patterns from the beginning of the program.
exit	causes the program to behave as if the end of the input had occurred (thus exit causes execution of the <i>END</i> statement if there is one)

Commenting

Comments in *awk* programs begin with the # character and end with the end of the line:

```
/string/ {print $2, $5} #Print fields 2 and 5 if string matches
```

Error Messages

Diagnostic output for *awk* is sparse and cryptic. Most *awk* errors that stop the program are syntax errors. Typical error statements are:

```
syntax error near line 1
illegal statement near line 1
```

Syntax errors often produce an additional message:

```
bailing out near line 1
```

meaning that the program gave up and returned control to the shell.

“Near” means that the line specified in the error message may not be the line that contains the actual error.

The message

```
funny variable xxxxxxxx
```

is *awk*'s response to a variable it can't deal with, such as a the negative field variable $\$(-1)$.

Some *awk* messages are more specific:

```
newline in character class near line 1
```

states the problem clearly.

Except when redirecting output, if you refer to a file that doesn't exist or can't be opened, you'll get the shell message:

```
awk: can't open file
```

Notes on the Design

Awk improves on *grep*, *egrep*, *fgrep*, *sed*, and *ed* by offering numeric processing, logical relations, and variables. *Awk* programs do not require compilation, as C programs do, and you do not need to know the C programming language to use *awk* (though it sometimes helps). *Awk* is one of the few tools on HP-UX that let you conveniently access fields within a line (*cut* is another such tool).

The designers of *awk* tried to integrate strings and numbers, treating all quantities as both, and postponing a choice until the last minute. This is why you can generally ignore the difference between a string and a number as you write a program.

Most *awk* users extract or manipulate information from the inputs. These usages are sometimes referred to as report generation and data transformation.

Notes on Awk Implementation

Aho, Weinberger, and Kernighan wrote *awk* using tools available on HP-UX, including *yacc* and *lex*. The elements that recognize regular expressions are deterministic finite automata, constructed directly from the expressions. When you invoke *awk*, your program is translated into a parse tree by the parser that was generated by *yacc* and *lex*. A simple interpreter executes the parse tree.

Awk is not fast. Breaking input into fields and delaying the evaluation of variable types are inherent bottlenecks. Further, there is no *awk* compiler, so you cannot use faster compiled versions of oft-used programs. The *awk command* is a machine that translates (parses) and interprets a program written in the *awk language* each time the program is run.

Annotated Examples

Generating Reports

One of the practical applications of *awk* is to put text into a different form or to alter its format for a particular requirement. This example shows how text can be selectively extracted and manipulated with *awk*.

The input file is a list of universities from the Big 8, Big 10, and Pac 10 athletic conferences. (If you wish to test this example, you must type in part of all of the file, or one like it.) The file lists the universities' names (one or two fields), the states in which they are located, and the seating capacities of their stadiums. The name of this file is *schools*. You can print the file with:

```
$ awk '{print $0}' schools
Arizona          AZ          Pac 10      52000
Arizona State    AZ          Pac 10      70021
Southern Cal     CA          Pac 10      92516
Stanford         CA          Pac 10      84892
UCLA             CA          Pac 10      92516
Washington       WA          Pac 10      59800
Washington State WA          Pac 10      40000
Oregon           OR          Pac 10      41009
Oregon State     OR          Pac 10      40593
California       CA          Pac 10      76780
Michigan         MI          Big 10      101701
Ohio State       OH          Big 10      85290
Indiana          IN          Big 10      52354
Iowa             IA          Big 10      66000
Illinois         IL          Big 10      70906
Michigan State   MI          Big 10      76000
Minnesota        MN          Big 10      62212
Northwestern     IL          Big 10      49256
Purdue           IN          Big 10      69250
Wisconsin        WI          Big 10      77280
Oklahoma         OK          Big 8       75008
Oklahoma State   OK          Big 8       50817
Missouri         MO          Big 8       62000
Iowa State       IA          Big 8       50000
Colorado         CO          Big 8       51805
Nebraska         NE          Big 8       73650
Kansas State     KS          Big 8       42000
Kansas           KS          Big 8       51500
$
```

Suppose you became interested in how many of these 28 schools (print out *NR* to verify that) were located in a particular state. Because each record contains the two-letter abbreviation of the school's state, the command:

```
$ awk '/CA/ {print $0}' schools
```

results in:

```
Southern Cal     CA          Pac 10      92516
Stanford         CA          Pac 10      84892
UCLA             CA          Pac 10      92516
California       CA          Pac 10      76780
```

Similarly, you can print all the records from a particular conference, or all the schools with “State” in their names:

```
$ awk '/State/ {print $0}' schools
Arizona State AZ Pac 10 70021
Washington State WA Pac 10 40000
Oregon State OR Pac 10 40593
Ohio State OH Big 10 85290
Michigan State MI Big 10 76000
Oklahoma State OK Big 8 50817
Iowa State IA Big 8 50000
Kansas State KS Big 8 42000
$
```

In the last two examples, pattern matching was done using regular expressions. This could have backfired if the file had included records on, say, the Central YMCA Community College of Chicago (first example) or of a college on Staten Island (second example). When designing patterns you should be aware of such potential pitfalls and think of ways around them.

Printing out all these records (lines) may not be all that you want to do. Suppose you're working for a professional soccer league that's looking for stadiums. You've been assigned to find stadiums with seating capacities greater than 60,000. The *schools* file is a limited resource, but it's a fair place to start. To find out which stadiums are big enough for your needs, use an action containing an **if** statement. Because you want to test every record, a pattern is not necessary. Type:

```
$ awk '{if ($NF) > 60000} print $0}' schools
```

and get:

```
Arizona State AZ Pac 10 70021
Southern Cal CA Pac 10 92516
Stanford CA Pac 10 84892
UCLA CA Pac 10 92516
California CA Pac 10 76780
Michigan MI Big 10 101701
Ohio State OH Big 10 85290
Iowa IA Big 10 66000
Illinois IL Big 10 70906
Michigan State MI Big 10 76000
Minnesota MN Big 10 62212
Purdue IN Big 10 69250
Wisconsin WI Big 10 77280
Oklahoma OK Big 8 75008
Missouri MO Big 8 62000
Nebraska NE Big 8 73650
```

Remember, *NF* is the predefined variable that means the number of fields in the current variable. You could do this job with just a pattern:

```
$ awk '$NF > 60000' schools
```

gets the same result (and it's shorter). If you want to save this information, type

```
$ awk '{if ($NF > 60000) print $0 >>big_stadiums}' schools
```

The `>>` operator appends the output to *big_stadiums*; you should use this operator as you search other files for similar information. Note the use of *\$NF* for the stadium-capacity field. This is used because the number of fields per record varies, but the stadium capacity is always in the last field.

Assume you want to find the average size of a group of stadiums. You need to scan all the records to add up the stadium sizes, then divide the total by the number of records to get the average. The final calculation takes place in an *END* statement after you have processed all of the input. Due to the size of this *awk* program, place it in a separate file:

```
$ cat >avg_capacity
{ x += $(NF) } # accumulate capacities in x
END { x /= NR; print "Average Capacity = ", x }
<control-d>
$
```

The `<control-d>` line represents typing **CONTROL-D**.

No quotes are needed around the program to protect it from shell interpretation, because you are not typing it on the command line. Also, you do not need to initialize *x*; *awk* sets *x* to the null string when it is created. The arithmetic is automatically done in floating point. To run the program, use the `-f` option:

```
$ awk -f avg_capacity schools
Average capacity = 64898.4
$
```

A longer program finds the average stadium capacity of each conference:

```
/Pac/      {xP += $(NF); iP++ }      #accum caps in xP; incr. count
/Big 10/   {x10 += $(NF); i10++}  #accum caps in x10; incr. count
/Big 8/    {x8 += $(NF); i8++}   #accum caps in x8; incr. count
END        {print "Avg. cap, Pac 10 = ", (xP/iP)
            print "Avg. cap, Big 10 = ", (x10/i10)
            print "Avg. cap, Big 8 = ", (x8/i8)
            }
```

The calculations and print statements for each conference cannot be on the same line with the accumulation statements because *awk* runs all the pattern-action pairs on each input record as it arrives. *END* is executed after all the input comes in. You don't know all the data until the *END* statement.

Notice that no semicolons are used in the multi-line *END* statement. *Awk* treats the newline as another expression separator.

The output of this program is

```
Avg. cap. Pac 10 = 65012,7
Avg. cap. Big 10 = 71024,9
Avg. cap. Big 8  = 57097,5
```

The information in the file *schools* forms a small database. *Awk* is useful for retrieving and manipulating information from such databases. Other applications include updating or reformatting input files.

One last job. Who has the biggest stadium?

```
$cat >findbiggest
{ x = $(NF); if (x>y) {y=x; bigrecord = $0}}
END {print bigrecord}
<control-d>
$

$ awk -f findbiggest schools
Michigan      MI      Big 10      101701
$
```

Doing Calculations

The following program finds the mean and the square root of the sum of the squares (root-mean-square or rms) of a list of input numbers. The work is performed on every record, so there are no patterns in this program other than *END*. This example is taken from “A Walk Through Awk,” a paper by Leon S. Levy of Bell Laboratories.

```
{sum_of_squares += $1 * $1} #accum sum of squares
{sum += $1} #accum nos for mean calc
END {mean = sum / NR #calc mean
      print "mean = ", mean
      rms = sqrt(sum_of_squares/NR) #calc rms
      print "rms = ", rms
}
```

Type this program into a file called *meanrms* and type some numbers into an input file, perhaps called *meanrms.data*.

```
$cat >meanrms.data
20
30
55
40
<control-d>
$ awk -f meanrms meanrms.data
mean = 36,25
rms = 38,487
$
```

The `<control-d>` notation means you should press **CONTROL-**D to end the new file.

You don't have to create an input file to use this program. You can run it using the standard input (your keyboard):

```
$ awk -f rms_mean -
```

The dash for the input is optional.

Type in each number you want in the calculation, pressing **RETURN** after each entry. After typing the last entry and pressing **RETURN**, type **CONTROL-**D (the end-of-file character). *Awk* then executes the *END* statement.

Because *awk* treats variables as strings until otherwise informed, giving character strings to this program produces unexpected answers. Mixing numbers and characters causes *awk* to calculate the mean and rms using the ASCII values of the characters.

Rearranging Data

You can use *awk* to change the format of records in a file. Assume you have this list of names in a file called *poets*.

```
Poe, Edgar Allan
Longfellow, Henry Wadsworth
Shakespeare, William
Frost, Robert
Dickinson, Emily
```

To transpose the first and last names, type

```
$ awk '{print $2, $1}' poets
```

This successfully switches around the first two fields, but it leaves out the middle names on two of the records and it leaves the commas on all the surnames.

To remove the commas, type

```
$ awk '{print $2, substr($1, 1, length($1)-1)}' poets
```

This complex-looking program drops the commas by printing a *substring* of the surname field consisting of everything *but* the comma, using the **substr** function. The first parameter, *\$1*, is the object of the **substr** function. The second parameter, *1*, means that we want the substring to start at the beginning of *\$1*. The third parameter, *length(\$1) - 1*, means that the substring should end one character before the end of *\$1* (just before the comma).

Saving the middle names is more awkward. Try adding *\$3* to the **print** action:

```
$ awk '{print $2, $3, substr($1, 1, length($1)-1)}' poets
```

The records with middle names come out correctly. But when no middle name is present an extra space occurs between the first and last names. The extra space is an output field separator being printed after *\$3*; when *\$3* is null (no middle name), you get two output field separators in a row.

One solution is to use an **if-else** statement to detect a middle name in the record. The program

```
{if (length($3) > 0) print $2, $3, substr($1, 1, length($1)-1)
else print $2, substr($1, 1, length($1)-1) }
```

when run:

```
$ awk -f reverse_names poets
```

results in:

```
Edgar Allan Poe
Henry Wadsworth Longfellow
William Shakespeare
Robert Frost
Emily Dickinson
```

Now that you have the format you want, save it by redirecting it to another file:

```
$ awk -f reverse_names poets >poets.awk
```

Another job you may want to do with a file like *poets* is to rearrange the records in alphabetical order. Unfortunately, *awk* cannot reorder records. Try using the *sort* utility. *Sort* is described in the *HP-UX Reference* and in the McGilton and Morgan book starting on page 138.

References

1. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. "Awk—A Pattern Scanning and Processing Language", Bell Laboratories, September 1978. Second Edition. (Not available from HP.)
2. Henry McGilton and Rachel Morgan. *Introducing the UNIX™ System*, McGraw Hill, 1983, pp. 177-184. (UNIX is a trademark of AT&T Bell Laboratories.) HP Part # 98680-90025.
3. *HP-UX Reference for the HP 9000 Series 200/500*. HP Part # 09000-90007.
4. Brian Kernighan and Dennis Ritchie. *The C Programming Language*, Prentice-Hall, 1978. HP part # 97089-90000.
5. "C Library Routines", Programming Environment, HP-UX Concepts and Tutorials.

