68NW9209H42A

SYSTEM V/88 Release 3.2
User's Guide

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 2565 PHOENIX, ARIZONA

POSTAGE WILL BE PAID BY ADDRESSEE

MOTOROLA INC.

Computer Group, Microcomputer Division, DW164
2900 South Diablo Way
Tempe, AZ 85282-9741

MOTOROL

Motorola welcomes your

Manual Title

Part Number

Your Name

Your Title

Company

Address

General Information:

Do you read this manu

☐ Install the product

☐ Reference inform

In general, how do you

☐ Index ☐ Table o

Completeness: ☐ Ex

What topic would you

Presentation: ☐ Excellent ☐ Very Good ☐ Good ☐ Fair ☐ Poor

What features of the manual are most useful (tables, figures, appendixes, index, etc.)?

Is the information easy to understand? ☐ Yes ☐ No If you checked no, please explain:

Is the information easy to find? ☐ Yes ☐ No If you checked no, please explain:

Technical Accuracy: ☐ Excellent ☐ Very Good ☐ Good ☐ Fair ☐ Poor

If you have found technical or typographical errors, please list them here.

Page Number	Description of Error

SYSTEM V/88 Release 3.2

User's Guide

(68NW9209H42A)

PREFACE

The material in this guide is organized into two major parts: an overview of the SYSTEM V/88 operating system and a set of tutorials covering the main tools available on the operating system. A brief description of each part follows. The last section of this introduction, *Notations and Conventions*, describes the typographical notations with which all the chapters of this guide conform. You may want to refer to that section from time to time as you read the guide.

First Edition February 1990

Contents

5 Line Editor Tutorial (ed)

6 Screen Editor Tutorial (vi)

Figures

Figures

Tables

1 What is the SYSTEM V/88 System?

System Overview

The first part of the SYSTEM V/88 User's Guide consists of Chapters 1 through 3, which introduce you to the basic principles of the operating system. Each chapter builds on information presented in preceding chapters, so it is important to read them in sequence.

- Chapter 1, *What is the SYSTEM V/88 System?*, provides an overview of the operating system.
- Chapter 2, *Basics for SYSTEM V/88 Users*, discusses the general rules and guidelines for using SYSTEM V/88. It covers topics related to using your terminal, obtaining a system account, and establishing contact with the operating system.
- Chapter 3, *Using the File System*, offers a working perspective of the file system. It introduces commands for building your own directory structure, accessing and manipulating the subdirectories and files you organize within it, and examining the contents of other directories in the system for which you have access permission.

SYSTEM V/88 Tutorials

The second part of this guide consists of tutorials on the following topics: the **ed** text editor, the **vi** text editor, the shell command and programming language, and electronic mail. For a thorough understanding of the material, we recommend that you work through the examples and exercises as you read each tutorial. (The tutorials assume you understand the concepts introduced in Chapters 1 through 3.)

- Chapter 4, *Overview of the Tutorials*, introduces four chapters of tutorials that cover command execution, text editing, electronic mail, elementary programming, and aids to software development.
- Chapter 5, *Line Editor Tutorial (ed)*, teaches you to how to use the **ed** text editor to create and modify text on a video display terminal or paper printing terminal.
- Chapter 6, *Screen Editor Tutorial (vi)*, teaches you how to use the visual text editor, **vi**, to create and modify text on a video display terminal.

NOTE

vi, the visual editor, is based on software developed by the Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California. Such software is owned and licensed by the Regents of the University of California.

- Chapter 7, *Shell Tutorial*, covers using the shell, both as a command interpreter and as a programming language for shell programs.
- Chapter 8, *Electronic Mail Tutorial*, teaches you how to send messages and files to users of both your SYSTEM V/88 and other SYSTEM V/88 systems.
- Chapter 9, *ksh Tutorial*, describes how to use the shell to write medium sized programming tasks at the shell level.

Features of SYSTEM V/88

The SYSTEM V/88 operating system is a set of programs (or software) that controls the computer, acts as the link between you and the computer, and provides tools to help you do your work. It provides an uncomplicated, efficient, and flexible computing environment. Specifically, the operating system offers these advantages:

- a general-purpose system for performing a wide variety of jobs or applications.
- an interactive environment that allows you to communicate directly with the computer and receive immediate responses to your requests and messages.

- a multi-user environment that allows you to share the computer's resources with other users without sacrificing productivity. This technique is called timesharing. The operating system interacts between users on a rotating basis so quickly that it appears to be interacting with all users simultaneously.
- a multitasking environment that enables you to execute more than one program simultaneously.

The organization of the operating system is based on four major components: the kernel, the file system, the shell, and the commands. The kernel is a program that constitutes the nucleus of the operating system; it coordinates the functioning of the computer's internals (e.g., allocating system resources). The kernel works invisibly; you need never be aware of it while doing your work. The file system provides a method of handling data that makes it easy to store and access information.

The shell is a program that serves as the command interpreter. It acts as a liaison between you and the kernel, interpreting and executing your commands. Because it reads input from you and sends you messages, it is described as interactive.

Commands are the names of programs that you request the computer to execute. Packages of programs are called tools. The operating system provides tools for jobs, e.g., creating and changing text, writing programs and developing software tools, exchanging information with others via the computer.

Reference Information

Several appendices and a glossary of operating system terms are also provided for reference.

- Appendix A, *Summary of the File System,* describes the directory structure and how information is stored in the operating system.
- Appendix B, *Summary of SYSTEM V/88 Commands,* lists in alphabetical order each command discussed in the guide.
- Appendix C, *Quick Reference to ed Commands*, summarizes the **ed** line editor commands organized by topic as in Chapter 5, *Line Editor Tutorial (ed).*

- Appendix D, *Quick Reference to vi Commands*, gives you a summary of **vi** commands, organized by topic as in Chapter 6, *Screen Editor Tutorial (vi)*.
- Appendix E, *Summary of Shell Command Language*, lists the vocabulary and programming constructs discussed in Chapter 7, *Shell Tutorial*.
- Appendix F, *Setting Terminal Type*, explains how to configure your terminal for use with the operating system.
- The Glossary defines those operating system terms that are used in this book.

Notations and Conventions

The following notations and conventions are used throughout this guide.

bold
User input (commands, options and arguments to commands, variables, and the names of directories and files) appears in **bold**.

italic
Names of variables to which values must be assigned (e.g., *password*) appear in *italic*.

`constant width`
Operating system output, e.g., prompts and responses to commands, appear in `constant width`.

<>
Input that does not appear on the screen when typed (passwords, tabs, and carriage returns) appears between angle brackets.

<CR>
represents the carriage return, i.e., the key marked RETURN.

<^char>
Control characters are shown between angle brackets because they do not appear on the screen when typed. The circumflex (^) represents the control key (labeled CTRL or `control` on your keyboard). To type a control character, hold down the CONTROL key while you type the character specified by *char*. For example, the notation <^**d**> means to hold down the CONTROL key while pressing the D key; the letter D does not appear on the screen.

[ ]
Command options and arguments that are optional, e.g., **[-msCj]**, are enclosed in square brackets.

|
The vertical bar separates optional arguments from which you may choose one. For example, when a command line has the following format:

command [*arg1* | *arg2*]

You may use either *arg1* or *arg2* when you issue the *command*.

...
Ellipses after an argument mean that more than one argument may be used on a single command line.

↑
Arrows on the screen (shown in examples in Chapter 6) represent the cursor.

command(number)
A command name followed by a number in parentheses refers to the part of a SYSTEM V/88 reference manual that documents that command: *User's Reference Manual*, *Programmer's Reference Manual*, or *System Administrator's Reference Manual*. For example, the notation **cat**(1) refers to the page in section 1 (of *User's Reference Manual*) that documents the **cat** command.

$
In sample commands, the **$** sign is used as the shell command prompt. This is not true for all systems. Whichever symbol your system uses, keep in mind that prompts are produced by the system; although a prompt is sometimes shown at the beginning of a command line as it would appear on your screen, you are not meant to type it. (The **$** sign is also used to reference the value of positional parameters and named variables; see Chapter 7 for details.)

In all chapters, full and partial screens are used to display examples of how your terminal screen will look when you interact with the operating system. These examples show how to use the operating system editors, write short programs, and execute commands. The input (characters typed by you) and output (characters printed by the operating system) are shown in these screens in accordance with the conventions listed above. All examples apply regardless of the type of terminal you use.

The commands discussed in each section of a chapter are reviewed at the end of that section. At the end of some sections, exercises are also provided so you can experiment with the commands. The answers to the exercises are at the end of the chapter.

NOTE

The text in this manual was prepared with the operating system text editors described in the guide and formatted with **troff**, **tbl**, **pic**, and **mm** macros.

SYSTEM V/88 Operating System Structure

Figure 1-1 is a model of the SYSTEM V/88 operating system. Each circle represents one of the main components of the operating system: the kernel, the shell, and user programs or commands. The arrows suggest the shell's role as the medium through which you and the kernel communicate. The remainder of this chapter describes each of these components, along with another important feature of the operating system, the file system.

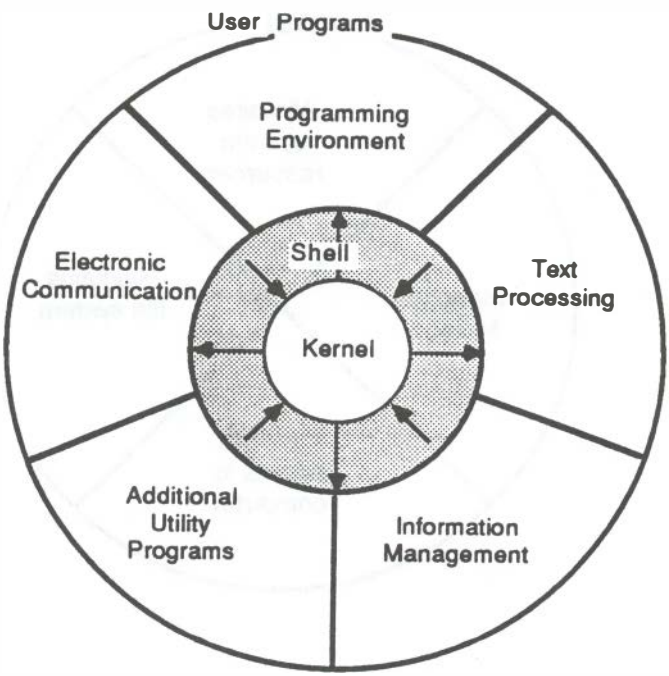


Figure 1-1. SYSTEM V/88 Operating System Model

The Kernel

The nucleus of the operating system is called the kernel. The kernel controls access to the computer, manages the computer's memory, maintains the file system, and allocates the computer's resources among users. Figure 1-2 is a functional view of the kernel.

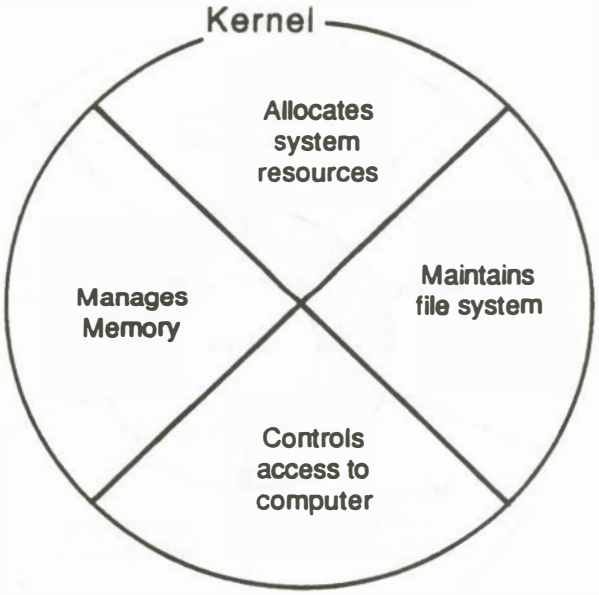


Figure 1-2. Functional View of the Kernel

The File System

The file system is the cornerstone of the operating system. It provides a logical method of organizing, retrieving, and managing information. The structure of the file system is hierarchical; it looks like an organization chart or an inverted tree (Figure 1-3).

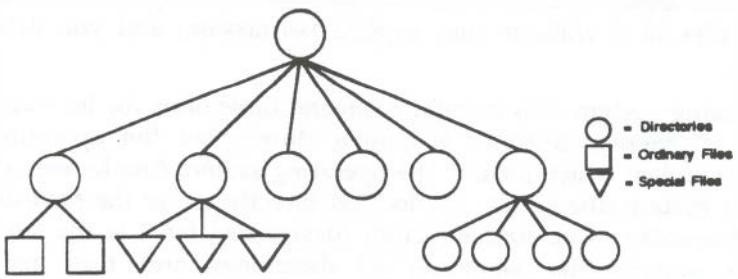


Figure 1-3. The Hierarchical Structure of the File System

The file is the basic unit of the operating system and can be any one of three types: an ordinary file, a directory, or a special file, see Chapter 3, *Using the File System.*

Ordinary Files

An ordinary file is a collection of characters that is treated as a unit by the operating system. Ordinary files are used to store any information you want to save. They may contain text for letters or reports, code for programs you write, or commands to run programs. Once you have created a file, you can add material to it, delete material from it, or remove it entirely when it is no longer needed.

Directories

A directory is a super-file that contains a group of related files. For example, a directory called **sales** may hold files containing monthly sales figures called **jan**, **feb**, **mar**. You can create directories, add or remove files from them, or remove directories at any time.

Generally, all directories you create and own are located in your home directory. This is a directory assigned to you by the system when you receive a recognized login. You have control over this directory; no one else can read or write files in it without your explicit permission, and you determine its structure.

The operating system also maintains several directories for its own use. The structure of these directories is similar throughout the operating system. These directories, which include the operating system (the kernel) and several important system directories, are located directly under the **root** directory in the file hierarchy. The **root** directory (designated by /) is the source of the operating system file structure; all directories and files are arranged hierarchically under it.

Special Files

Special files constitute the most unusual feature of the file system. A special file represents a physical device such as a terminal, disk drive, magnetic tape drive, or communication link. The system reads and writes to special files in the same way it does to ordinary files. However the system's read and write requests do not activate the normal file access mechanism; instead, they activate the device handler associated with the file.

Some operating systems require you to define the type of file you have and to use it in a specified way. In those cases, you must consider how the files are stored since they might be sequential, random-access, or binary files. To SYSTEM V/88 however, all files are alike. This makes the operating system file structure easy to use. For example, you need not specify memory requirements for your files since the system automatically does this for you. Or a program you write needs to access a certain device, e.g., a printer, specify the device just as you would another one of your files. In the operating system, there is only one interface for all input from you and output to you; this simplifies your interaction with the system.

Figure 1-4 shows an example of a typical file system. Notice that the root directory contains the kernel (**/unix**) and several important system directories.

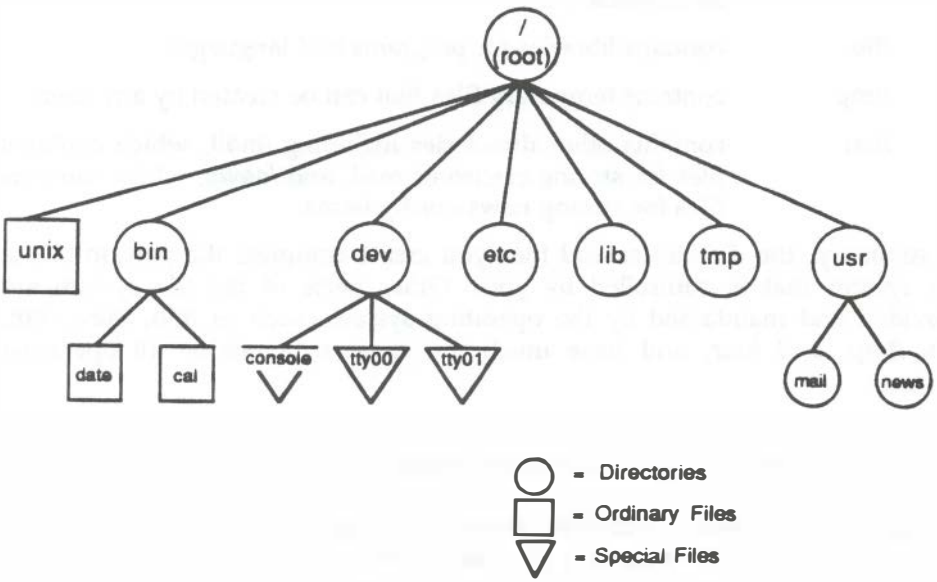


Figure 1-4. Example of a File System

Directory	Contents
/bin	contains many executable programs and utilities.
/dev	contains special files that represent peripheral devices, e.g., the console, line printer, user terminals, and disks.
/etc	contains programs and data files for system administration.
/lib	contains libraries for programs and languages.
/tmp	contains temporary files that can be created by any user.
/usr	contains other directories including **/mail**, which contains files for storing electronic mail, and **/news**, which contains files for storing newsworthy items.

In summary, the directories and files you create comprise the portion of the file system that is controlled by you. Other parts of the file system are provided and maintained by the operating system, such as **/bin, /dev, /etc, /lib, /tmp**, and **/usr**, and have much the same structure on all operating systems.

You will learn more about the file system in other chapters. Chapter 3 shows how to organize a file system directory structure, and access and manipulate files. Chapter 4 gives an overview of the operating system capabilities. The effective use of these capabilities depends on your familiarity with the file system and your ability to access information stored within it. Chapters 5 and 6 are tutorials designed to teach you how to create and edit files.

The Shell

The shell is a unique command interpreter that allows you to communicate with the operating system. The shell reads the commands you enter and interprets them as requests to execute other programs, access files, or provide output. The shell is also a powerful programming language, not unlike the C programming language, that provides conditional execution and control flow features. The model of an operating system in Figure 1-1 shows the two-way flow of communication between you and the computer via the shell.

Chapter 4 describes the shell's capabilities. Chapter 7 is a tutorial that teaches you to write simple shell programs called shell scripts and tailor your environment.

Commands

A program is a set of instructions to the computer. Programs that can be executed by the computer without need for translation are called executable programs or commands. As a typical user of the operating system, you have many standard programs and tools available to you. If you use the operating system to write programs and develop software, you can also draw on system calls, subroutines, and other tools. Of course, any programs you write yourself are also at your disposal.

This book introduces you to many operating system programs and tools that you will use on a regular basis. If you need additional information on these or other standard programs, refer to the *User's Reference Manual.* For information on tools and routines related to programming and software development, consult the *Programmer's Reference Manual.*

The reference manuals may also be available online. (Online documents are stored in your computer's file system.) You can summon pages from the online manuals by executing the command **man** (short for manual page). For details on using the **man** command, refer to the **man**(1) page in the *User's Reference Manual.*

What Commands Do

The outer circle of the operating system model in Figure 1-1 organizes the system programs and tools into functional categories. These functions include:

text processing
: The system provides programs such as line and screen editors to create and change text, a spell checker for locating spelling errors, and optional text formatters to produce high-quality paper copies that are suitable for publication.

information management
: The system provides many programs that allow you to create, organize, and remove files and directories.

electronic communication
Several programs, e.g., **mail**, enable you to transmit information to other users and to other operating systems.

software development
Several operating system programs establish a friendly programming environment by providing operating-system-to-programming language interfaces and by supplying numerous utility programs.

additional utilities
The system also offers capabilities to generate graphics and perform calculations.

Command Syntax

To make your requests comprehensible to the operating system, you must present each command in the correct format, or command line syntax. This syntax defines the order in which you enter the components of a command line. Just as you must put the subject of a sentence before the verb in an English sentence, so must you put the parts of a command line in the order required by the command line syntax. Otherwise, the operating system shell will not be able to interpret your request. Here is a typical example of the syntax of an operating system command line:

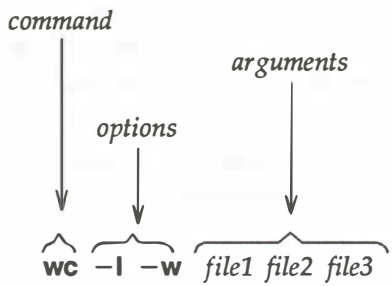
command *option(s)* *argument(s)*<**CR**>

On every operating system command line, you must type at least two components: a command name and the **RETURN** key. (The notation <**CR**> is used as an instruction to press the **RETURN** key throughout this guide.) A command line may also contain either options or arguments, or both. What are commands, options, and arguments?

- A *command* is the name of the program you want to run.
- An *option* modifies how the command runs.
- An *argument* specifies data on which the command is to operate (usually the name of a directory or file).

In command lines that include options and arguments, the component words are separated by at least one blank space; insert a blank by pressing the space bar. If an argument name contains a blank, enclose that name in quotation marks. For example, if the argument to your command is **sample 1**, you must type it as "**sample 1**". If you forget the quotation marks, the shell will interpret **sample** and **1** as two separate arguments.

Some commands allow you to specify several options and arguments on a command line. Consider the following command line:



In this example, **wc** is the name of the command and two options, **-l** and **-w**, have been specified. (The operating system usually allows you to group options such as these to read **-lw** if you prefer.) In addition, three files (*file1*, *file2*, and *file3*) are specified as arguments. Although most options can be grouped together, arguments cannot.

The following examples show the proper sequence and spacing in command line syntax:

Incorrect	Correct
wc*file*	**wc** *file*
wc−l*file*	**wc** **−l** *file*
wc **−l** **w** *file*	**wc** **−lw** *file*
	or
	wc **−l** **−w** *file*
wc *file1file2*	**wc** *file1* *file2*

Remember, regardless of the number of components, you must end every command line by pressing the **RETURN** key.

Command Execution

Figure 1-5 shows the flow of control when SYSTEM V/88 executes a command.

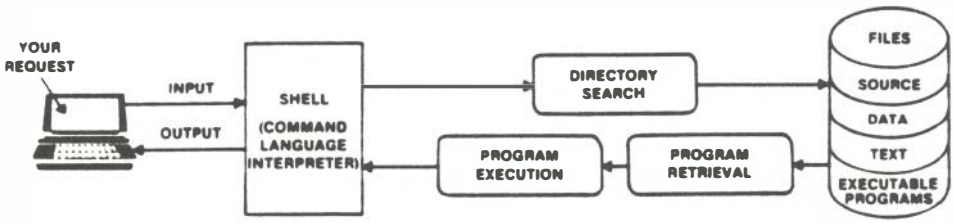


Figure 1-5. Execution of an Operating System Command

To execute a command, enter a command line when a prompt (e.g., $ sign) appears on your screen. The shell considers your command as input, searches through one or more directories to retrieve the program you specified, and conveys your request with the program requested to the kernel. The kernel then follows the instructions in the program and executes the command you requested. After the program has finished running, the shell signals that it is ready for your next command by displaying another prompt.

This chapter has described some basic principles of the operating system. The following chapters will help you apply these principles according to your computing needs.

2 Basics for SYSTEM V/88 Users

Getting Started

This chapter acquaints you with the general rules and guidelines for getting started on SYSTEM V/88. Specifically, it lists terminal settings and explains how to use the keyboard, obtain a login, log on and off the system, and enter simple commands.

To establish contact with the operating system, you need:

- a terminal
- a login name (a name by which the operating system identifies you as one of its authorized users)
- a password that verifies your identity
- instructions for dialing in and accessing the operating system when your terminal is not directly connected to the computer

This chapter follows the notations and conventions used throughout this guide. For a description, see *What is the SYSTEM V/88 System.*

The Terminal

A terminal is an input/output device: you use it to enter requests to the operating system; the system uses it to send its responses to you. There are two basic types of terminals: video display terminals and printing terminals.

The video display terminal shows its input and output on a display screen; the printing terminal, on continuously fed paper. Instructions throughout this book that refer to the terminal screen apply also to the terminal screen.

Required Terminal Settings

Regardless of the type of terminal you use, you must configure it properly to communicate with the operating system. If you have not set terminal options before, seek help from someone who has done this function.

How you configure a terminal depends on the type of terminal you are using. Some terminals are set with switches; others are set directly from the keyboard by using function keys. To determine how to make the settings on your terminal, consult the owner's manual provided by the manufacturer.

The following is a list of configuration checks you should do on any terminal before trying to log in on the operating system:

1. Turn on the power.
2. Set the terminal to ONLINE or REMOTE operation. This setting ensures the terminal is under the direct control of the computer.
3. Set the terminal to FULL DUPLEX mode. This mode ensures two-way communication (input/output) between you and the operating system.
4. If your terminal is not directly connected (hard-wired) to the computer, make sure the acoustic coupler or data-phone set you are using is set to the FULL DUPLEX mode.
5. Set character generation to LOWERCASE. If your terminal generates only uppercase letters, the operating system will accommodate it by printing everything in uppercase letters.
6. Set the terminal to NO PARITY.
7. Set the baud rate to 9600 (or whatever speed the System Administrator assigned to the port). This is the speed at which the computer communicates with the terminal, measured in bits-per-second (bps). For example, a terminal set at a baud rate of 9600 sends and receives 960 *characters* per second. Depending on the computer and terminal, baud rates between 300 and 19200 are available.

Keyboard Characteristics

There is no standard layout for terminal keyboards. However, all terminal keyboards share a standard set of 128 characters called the ASCII character set. (ASCII is an acronym for American Standard Code for Information Interchange.) While the keys are labeled with characters that are meaningful to you (e.g., the letters of the alphabet), each one is also associated with an ASCII code that is meaningful to the computer. Figure 2-1 shows an example of a keyboard on an ASCII terminal.

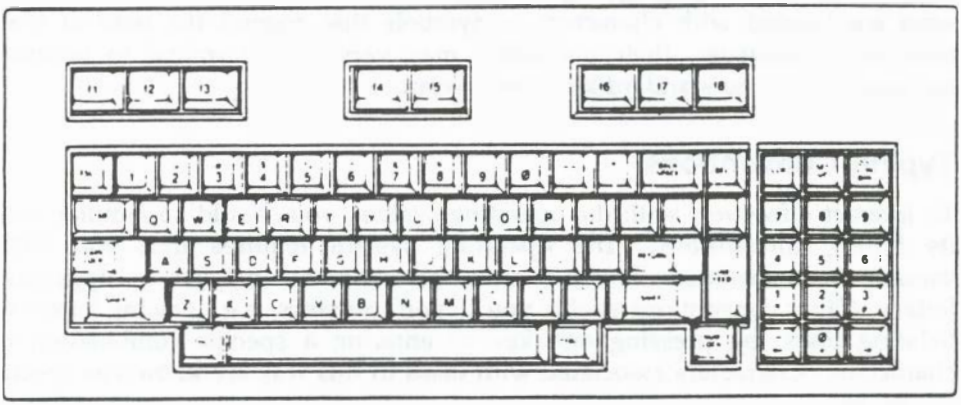


Figure 2-1. Keyboard Layout of a Typical Terminal

The keyboard layout on a typical ASCII terminal is basically the same as a typewriter's, with a few additional keys for functions such as interrupting tasks:

- the letters of the English alphabet (both uppercase and lowercase)
- the numerals (0 through 9)
- a set of symbols including ! @ # $ % ^ & ( ) _ - + = ~ ' { } [ ] \ : ; " ' < > , ? /
- specially defined words e.g., **RETURN** and **BREAK** and abbreviations e.g., **DEL** for delete, **CTRL** for control, and **ESC** for escape.

While terminal and typewriter keyboards both have alphanumeric keys, terminal keyboards also have keys designed for use with a computer. These keys are labeled with characters or symbols that remind the user of their functions. However, their placement may vary from terminal to terminal because there is no standard keyboard layout.

Typing Conventions

To interact effectively with the operating system, you should be familiar with its typing conventions. The operating system requires that you enter commands in lowercase letters (unless the command includes an uppercase letter). Other conventions enable you to perform tasks, e.g., erasing letters or deleting lines, by pressing one key or entering a specific combination of characters. Characters associated with tasks in this way are known as special characters.

Table 2-1 lists the conventions based on special characters. They are explained in detail in the following sections.

Command Prompt

The standard operating system command prompt is the dollar sign (**$**). When it appears on your terminal screen, the operating system is waiting for instructions from you. The appropriate response to the prompt is to issue a command and press the **RETURN** key.

The **$** is the default value for the command prompt. Chapter 7 explains how to change it when you prefer another character or character string as your command prompt.

Table 2-1. Special Characters and Symbols

Keys	Meaning
$	System's command prompt (your cue to issue a command).
<CERASE>	Erase a character.
<CKILL>	Erase or kill an entire line.
<BREAK>	Stop execution of a program or command. <DEL> Delete or kill the current command line.
<ESC>	When used with another character, performs a specific function (called an escape sequence). When used in an editing session with the **vi** editor, ends the text input mode and returns you to the command mode.
<CR>	End a line of typing and put the cursor on a new line. To type <CR>, press the RETURN key.
<^d>	Stop input to the system or log off.
<^h>	Backspace for terminals without a backspace key.
<^i>	Horizontal tab for terminals without a tab key.
<^s>	Temporarily stops output from printing on the screen.
<^q>	Makes the output resume printing on the screen after it has been stopped by the **<^s>** command.

NOTE: Non printing characters are shown in angle brackets (<>). Characters preceded by a circumflex (^) are called control characters and are pronounced control-*letter*. To type a control character, hold down the CTRL key and press the specified letter.

Correcting Typing Errors

There are two keys you can use to delete text so that you can correct typing errors. For the remainder of the guide, they are referred to as **CKILL** which kills the current line, and **CERASE**, which erases the last character typed. By default, unless otherwise changed directly via **stty**(1), or indirectly (by System Administrator) through the use of the Terminal Support programs (see Appendix F). The assignments of these keys are:

CERASE = # (pound sign)
CKILL = @ (at)

For information or instructions on reassigning these functions to other keys, see Appendix F and *Reassigning the Delete Functions* in this section and Chapter 7.

If **CERASE** and/or **CKILL** have been reassigned, you normally receive a reminder of the current values when you log in. At any time, you may execute the **stty** command (without arguments) that lists the current erase and kill characters if they are different from the defaults.

NOTE

There are some cases where these keys cannot be assigned, e.g., when logging on to the system.

In addition, a third key is available that can kill the current command line, i.e., when interacting directly with the shell or the editor. This is referred to as **CINTR** because it also functions as an **interrupt** key (see *Stopping a Command*) later in this section.

Deleting the Current Line: CKILL Character

The **CKILL** character kills the current line. When you enter it, the cursor moves to the next line. The line containing the error is not erased from the screen but is ignored. If **CKILL** is a printable character, it is added to the end of the line before the cursor moves down.

The **CKILL** character works only on the current line; be sure to enter it before you press the RETURN key if you want to kill a line. In the following example, a misspelled command is typed on a command line; the command is cancelled with the @ key (default **CKILL**):

whooo@
who<CR>

Deleting the Last Characters Typed: CERASE Character

The **CERASE** character deletes the characters last typed on the current line. When you type **CERASE**, the cursor backs up over the last character and erases it, letting you retype it. This is an easy way to correct a typing error.

You can delete as many characters as you like as long as you type a corresponding number of **CERASE** characters. For example, in the following command line, two characters are deleted by typing two # signs (default **CERASE**):

dattw##e<CR>

The operating system does not normally print the "#", but instead, visually erases the previous character.

NOTE

The above system behavior is controlled somewhat by the **stty echoe** and **stty -echoe** commands. The above descriptions assume **stty echoe** is in effect. If not, the cursor does not move backward unless **CERASE** = BACKSPACE, in which case a character is not visually erased until a new character is typed.

Reassigning the Delete Functions

As stated earlier, you can change the keys that kill lines and erase characters. If you want to change these keys for a single working session, you can issue a command to the shell to reassign them; the delete functions will revert to the default keys (# and @) as soon as you log off. If you want to use other keys regularly, you must specify the reassignment in a file called **.profile**. Instructions for making both temporary and permanent key reassignments, and a description of the **.profile**, are given in Chapter 7.

There are three points to keep in mind when you reassign the delete functions to non-default keys. First, the operating system allows only one key at a time to perform a delete function. When you reassign a function to a non-default key, you also take that function away from the default key. For example, if you reassign the erase function from the # key to the BACKSPACE key, you will no longer be able to use the # key to erase characters. Neither will you have two keys that perform the same function.

Secondly, such reassignments are inherited by any other operating system program that allows you to perform the function you have reassigned. For example, the interactive text editor **ed** (described in Chapter 5) allows you to delete text with the same key you use to correct errors on a shell command line (as described in this section). Therefore, if you reassign the erase function to the BACKSPACE key, you will have to use the BACKSPACE key to erase characters while working in the **ed** editor, as well. The # key will no longer work.

Finally, keep in mind that any reassignments you have specified in your **.profile** do not become effective until after you log in. Therefore, if you make an error while typing your login name or password, you must use the # or @ key to correct it.

Whichever keys you use, remember that they work only on the current line. Be sure to correct your errors before pressing the RETURN key at the end of a line.

Using Special Characters as Literal Characters

What happens if you want to use a special character with literal meaning as a unit of text? Since the operating system default behavior is to interpret special characters as commands, you must tell the system to ignore or escape from a character's special meaning whenever you want to use it as a literal character. The backslash (\) enables you to do this. Type a \ before any special character that you want to have treated as it appears. By doing this you essentially tell the system to ignore this character's special meaning and treat it as a literal unit of text.

For example, you want to add the following sentence to a file and **CERASE** = #:

> Only one # appears on this sheet of music.

To prevent the operating system from interpreting the # key as a request to delete a character, enter a \ in front of the #. If you do not, the system will erase the space after the word one and print your sentence as follows:

```
Only one appears on this sheet of music.
```

To avoid this, type your sentence as follows:

> **Only one \# appears on this sheet of music.**

Typing Speed

After the prompt appears on your terminal screen, you can type as fast as you want, even when the operating system is executing a command or responding to one. Since your input and the system's output appear on the screen simultaneously, the printout on your screen will appear garbled. However, while this may be inconvenient for you, it does not interfere with the operating system's work because it has read-ahead capability. This capability allows the system to handle input and output separately. The system takes and stores input (your next request) while it sends output (its response to your last request) to the screen.

Stopping a Command

If you want to stop the execution of a command, press the **BREAK** or enter the **CINTR** character. The operating system stops the program and prints a prompt on the screen. This is its signal that it has stopped the last command from running and is ready for your next command. By default, the following assignment exists:

CINTR = **<DEL>** (Delete Key)

See Appendix F for instruction on how to reassign this character.

NOTE

On some terminals, the **DELETE** or **BREAK** keys may have another name.

Using Control Characters

Locate the control key on your terminal keyboard. It may be labeled **CONTROL** or **CTRL** and is probably to the left of the A key or below the Z key. The **CTRL** key is used in combination with other characters to perform physical controlling actions on lines of typing. Commands entered in this way are called control characters. Some control characters perform mundane tasks such as backspacing and tabbing. Others define commands that are specific to the operating system. For example, one control character (**CTRL-S**) temporarily halts output that is being printed on a terminal screen.

To type a control character, hold down the **CTRL** key and press the appropriate alphabetic key. Most control characters do not appear on the screen when typed and therefore are shown between angle brackets (see *Notations and Conventions* in Chapter 1, *What is the SYSTEM V/88 System?*). The **CTRL** key is represented by a circumflex (^) before the letter; **<^s>** designates the **CTRL-S** character.

The two functions for which control characters are most often used are to control the printing of output on the screen and to log off the system. To prevent information from rolling off the screen on a video display terminal, type **<^s>**; the printing stops. When you are ready to read more output, type **<^q>** and the printing resumes.

To log off the operating system, type <^**d**>. (See *Logging Off* later in this chapter for a detailed description of this procedure.)

In addition, the operating system uses control characters to provide capabilities that some terminals fail to make available through function specific keys. If your keyboard does not have a **BACKSPACE** key, you can use the <^**h**> key instead. You can also set tabs without a tab key by typing <^**i**> if your terminal is set properly. (Refer to the section entitled *Possible Login Problems* for information on setting the tab key.)

Now that you have configured the terminal and inspected the keyboard, one step remains before you can establish communication with the operating system: you must obtain a login name.

Obtaining a Login Name

A login name is the name by which the operating system verifies that you are an authorized user of the system when you request access to it. It is so called because you must enter it every time you want to log in. (The expression *logging in* is derived from the fact that the system maintains a log for each user, in which it records the type and amount of system resources being used.)

To obtain a login name, set up an operating system account through your local System Administrator. There are few rules governing your choice of a login name. Typically, it is three to eight characters long. It can contain any combination of lowercase alphanumeric characters, as long as it starts with a letter. It cannot contain any symbols.

However, your login name is probably determined by local practices. The users of your system may all use their initials, last names, or nicknames as their login names. Here are a few examples of legal login names: **starship**, **mary2**, and **jmrs**.

Establishing Contact

Typically, you will be using either a terminal that is wired directly to a computer or a terminal that communicates with a computer over a telephone line.

NOTE

This section describes a typical procedure for logging in that may not apply to your system. There are many ways to log in on an operating system over a telephone line. Security precautions on your system may require that you use a special telephone number or other security code. For instructions on logging in your operating system from outside your computer installation site, see your System Administrator.

Turn on your terminal. If it is directly connected, the `login:` prompt immediately appears in the upper left-hand corner of the screen.

If you are going to communicate with the computer over a telephone line, you must now establish a connection. The following procedure is an example of one method you might use. (For the procedure used on your system, see your System Administrator.)

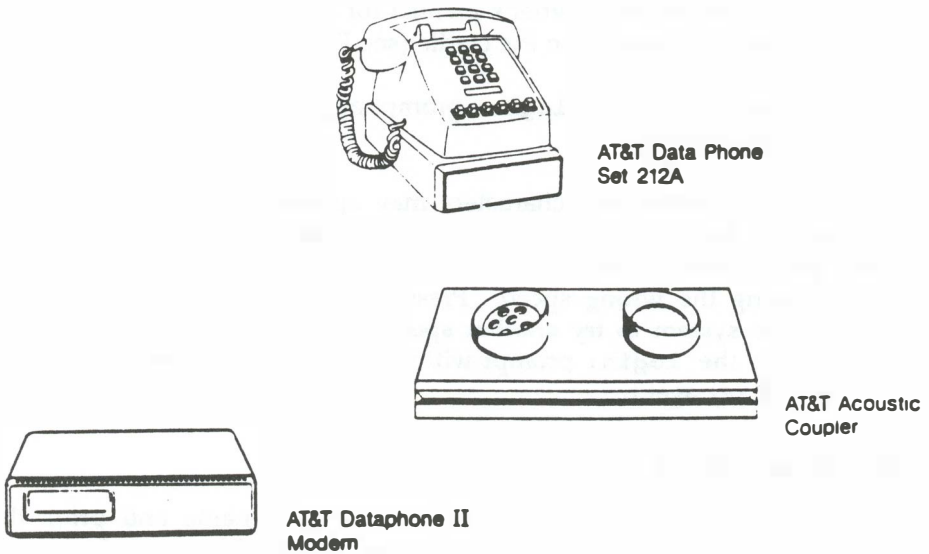


Figure 2-2. Data Phone Set, Modem, and Acoustic Coupler

1. Dial the telephone number that connects you to the operating system. You will hear one of the following:
   - A high-pitched tone. This means that the system is accessible.
   - A busy signal. This means that either the circuits are busy or the line is in use. Hang up and dial again.
   - Continuous ringing and no answer. This usually means that there is trouble with the telephone line or that the system is inoperable because of mechanical failure or electronic problems. Hang up and dial again later.

2. When you hear the high-pitched tone, place the handset of the phone in the acoustic coupler or momentarily press the appropriate button on the data phone set (see the owner's manual for the appropriate equipment). Then replace the handset in the cradle (see Figure 2-2).

3. After a few seconds, the `login:` prompt appears in the upper left-hand corner of the screen.

4. A series of meaningless characters may appear on your screen. This means that the telephone number you called serves more than one baud rate; the operating system is trying to communicate with your terminal, but is using the wrong speed. Press the BREAK or RETURN key; this signals the system to try another speed. If the operating system does not display the `login:` prompt within a few seconds, press the BREAK or RETURN key again.

Login Procedure

When the `login:` prompt appears, type your login name and press the RETURN key. For example, if your login name is **starship**, your login line appears as:

`login:` **starship<CR>**

NOTE

Remember to type in lowercase letters. If you use uppercase from the time you log in, the operating system expects and responds in uppercase exclusively until the next time you log in. It accepts and runs many commands typed in uppercase, but will not allow you to edit files.

Entering Your Password

After you have typed your login name correctly, the system prompts you for your password. Type your password and press the **RETURN** key. For security reasons, the operating system does not print (or echo) your password on the screen.

When both your login name and password are acceptable to the operating system, it prints the message of the day, current news items, and the default command prompt (**$**). (The message of the day might include a schedule for system maintenance, and news items might include an announcement of a new system tool.) When you have logged in, your screen appears similar to the following:

```
login: starship<CR>
password:
SYSTEM V/88 system news
$
```

If you make a typing mistake when logging in, the operating system prints the message **login incorrect** on your screen. Then it gives you a second chance to log in by printing another **login:** prompt:

```
login: starship<CR>
password:
login incorrect
login:
```

The login procedure may also fail if the communication link between your terminal and the operating system has been dropped. If this happens, you must reestablish contact with the computer (specifically, with the data switch that links your terminal to the computer) before trying to log in again. Since procedures for doing this vary from site to site, ask your System Administrator to give you exact instructions for getting a connection on the data switch.

If you have never logged in on the operating system, your login procedure may differ from the one just described. This is because some System Administrators follow the optional security procedure of assigning temporary passwords to new users when they set up their accounts. If you have a temporary password, the system will force you to choose a new password before it allows you to log in.

By forcing you to choose a password for your exclusive use, this extra step helps to ensure a system's security. Protection of system resources and your personal files depends on your keeping your password private.

The actual procedure you follow is determined by the administrative procedures at your computer installation site. However, it will probably be similar to the following example of a first-time login procedure:

1. You establish contact; the operating system displays the `login:` prompt. Type your login name and press the **RETURN** key.
2. The operating system prints the password prompt. Type your temporary password and press the **RETURN** key.
3. The system tells you your temporary password has expired and you must select a new one.
4. The system asks you to type your old password again. Type your temporary password.
5. The system prompts you to type your new password. Type the password you have chosen.

   Passwords must be constructed to meet the following requirements:

   - Each password must have at least six characters. Only the first eight characters are significant.
   - Each password must contain at least two alphabetic characters and at least one numeric or special character. Alphabetic characters can be uppercase or lowercase letters.

- Each password must differ from your login name and any reverse or circular shift of that login name. For comparison purposes, an uppercase letter and its corresponding lowercase letter are equivalent.
- A new password must differ from the old by at least three characters. In this comparison, an uppercase letter and its corresponding lowercase letter are equivalent.

Examples of valid passwords: **mar84ch**, **Jonath0n**, and **BRAV3S**.

NOTE

The operating system you are using may have different requirements to consider when choosing a password. Ask your System Administrator for details.

6. For verification, the system asks you to re-enter your new password. Type your new password again.
7. If you do not re-enter the new password exactly as typed the first time, the system tells you the passwords do not match and asks you to try the procedure again. On some systems, however, the communication link may be dropped if you do not re-enter the password exactly as typed the first time. If this happens, you must return to step 1 and begin the login procedure again. When the passwords match, the system displays the prompt.

The following screen depicts this procedure (steps 1 through 6) for first-time operating system users.

```
login: starship <CR>
password: <CR>
Your password has expired.
Choose a new one.
Old password: <CR>
New password: <CR>
Re-enter new password: <CR>
$
```

Possible Login Problems

A terminal usually behaves predictably when you have made the appropriate terminal settings. Sometimes, however, it may act peculiarly. For example, the carriage return may not work properly.

Some problems can be corrected by logging off the system and logging in again. If logging in a second time does not remedy the problem, you should first check the following and try logging in once again:

keyboard
: Keys labeled **CAPS**, **LOCAL**, **BLOCK**, and so on should not be enabled (put into the locked position). You can usually disable these keys simply by pressing them.

data phone set or modem
: If your terminal is connected to the computer via telephone lines, check the baud rate and duplex settings.

switches
: Some terminals have several switches that must be set. Check that all are set properly.

Numerous problems can occur if your terminal is not set properly. To verify the terminal settings, check the settings listed under *Required Terminal Settings.* If you need additional information about the keyboard, terminal, data phone, or modem, check the owner's manual for the appropriate equipment.

Table 2-2 presents suggestions for detecting, diagnosing, and correcting some problems you may experience when logging in. If you need further help, contact your System Administrator.

Table 2-2. Troubleshooting Login Problems

Problem	Possible Cause	Action/Remedy
Meaningless characters	SYSTEM V/88 at wrong speed	Press RETURN or BREAK key
Input and output appear in uppercase letters	Terminal has been set to uppercase setting	Log off and set character generation to lower case
Input appears in uppercase, output in lower case	Key labeled CAPS (or CAPS LOCK) is enabled	Press CAPS or CAPS LOCK key to disable setting
Input is printed twice	Terminal is set to HALF DUPLEX	Change setting to FULL DUPLEX
Tab key does not work properly	Tabs are not set correctly	Type **stty –tabs** (current session only†).
Communication link cannot be established although high pitched tone is heard when dialing in	Terminal is set to LOCAL or OFF-LINE	Set terminal to ONLINE try logging in again
Communication link (terminal to SYSTEM V/88) is repeatedly dropped	Bad telephone line or bad communications port	Call System Administrator

† To ensure a correct tab setting for all sessions, add the line **stty –tabs** to your **.profile** file (see Chapter 7).

Simple Commands

When the prompt appears on your screen, the operating system has recognized you as an authorized user and is waiting for you to request a program by entering a command. For example, try running the **date** command. After the prompt, type the command and press the RETURN key.

The operating system accesses a program called **date**, executes it, and prints its results on the screen:

```
$ date<CR>
Wed Oct 15 09:49:44 EDT 1986
$
```

The **date** command prints the date and time, using the 24-hour clock. Now type the **who** command and press the **RETURN** key. Your screen appears similar to the following:

```
$ who<CR>
starship      tty00      Oct 12     8:53
mary2         tty02      Oct 12     8:56
acct123       tty05      Oct 12     8:54
jmrs          tty06      Oct 12     8:56
$
```

The **who** command lists the login names of everyone currently working on your system. The **tty** designations refer to the special files that correspond to each user's terminal. The date and time at which each user logged in are also shown.

The help Command

To help you learn how to use these and other commands, the operating system provides an online teaching aid: the **help** command. This program tells you which command you need to perform a particular task and how to execute specific commands. For novice users of the operating system, it also provides definitions of vocabulary and explanations of basic concepts about the system.

NOTE

> The **help** command is not available on all operating systems; check with your System Administrator to find out if it is installed on your system.

When you need assistance, type **help** and press the RETURN key. The program gives you a choice of four ways in which it can help you: by providing general information; locating the appropriate command for a particular task; giving you instructions on how to use a particular command; and defining terms. The following example shows how this menu appears on your screen when you type the command.

```
$ help<CR>
help:   UNIX System On-Line Help


choices         description
s               starter: general information

l               locate: find a command with keywords

u               usage: information about commands

g               glossary: definitions of terms


r               Redirect to a file or a command
q               Quit


Enter choice > _
```

Each item under `description` on this menu (**starter**, **locate**, **usage**, and **glossary**) is an interactive menu program. Request one of these programs by typing the option listed beside it under `choices` (e.g., **u**).

Because **starter**, **locate**, **usage**, and **glossary** are programs, they can also be called from the shell. Once you are familiar with them, you can skip the step of entering the **help** command first. If you know which program you want to run, you can call it by typing its name as either a command or an argument to the **help** command. For example, to call the **usage** program, use one of the following command lines:

help usage<CR>

or

usage<CR>

The program you choose responds by printing a summary of its function, a menu of choices, instructions, and examples of how to follow the instructions. In this way, the **help** program leads you through a series of steps that enable you to get the information you need.

Logging Off

When you have completed a session with the operating system, type <^**d**> after the prompt. (Remember that control characters such as <^**d**> are typed by holding down the CTRL key and pressing the appropriate alphabetic key. Because they are nonprinting characters, they do not appear on your screen.) After several seconds, the operating system displays the `login:` prompt again:

```
$ <^d>
login:
```

This shows that you have logged off successfully and the system is ready for someone else to log in.

NOTE

Always log off the operating system by typing <^**d**> before you turn off the terminal or hang up the telephone. If you do not, you may not be actually logged off the system.

The **exit** command also allows you to log off but is not used by most users. It may be convenient if you want to include a command to log off within a shell program. (For details, refer to the *Special Commands* section of the **sh**(1) page in the *User's Reference Manual*.)

3 Using the File System

Introduction

To use the SYSTEM V/88 file system effectively, you must be familiar with its structure, know something about your relationship to this structure, and understand how the relationship changes as you move around within it. This chapter prepares you to use this file system.

The first two sections (*File System Structure* and *Your Place in the File System*) offer a working perspective of the file system. The remaining sections describe operating system commands that allow you to build your own directory structure, manipulate the subdirectories and files you organize within it, and examine the contents of other directories in the system for which you have access permission.

Each command is discussed in a separate subsection. Tables at the end of these subsections summarize the features of each command so that you can later review a command's syntax and capabilities quickly. Many of the commands presented in this section have additional, sophisticated uses. These, however, are left for more experienced users and are described in other SYSTEM V/88 documentation. All the commands presented here are basic to using the file system efficiently and easily. Try using each command as you read about it.

File System Structure

The file system is made up of a set of ordinary files, special files, and directories. These components provide a way to organize, retrieve, and manage information electronically. Chapter 1 introduced the properties of directories and files; this section reviews them briefly before discussing how to use them.

- An ordinary file is a collection of characters stored on a disk. It may contain text for a report or code for a program.
- A special file represents a physical device, e.g., a terminal or disk.
- A directory is a collection of files and other directories (sometimes called subdirectories). Use directories to group files together on the basis of any criteria you choose. For example, you might create a directory for each product that your company sells or for each of your student's records.

The directories and files are organized into a tree-shaped structure. Figure 3-1 shows a sample file structure with a directory called **root** (/) as its source. By moving down the branches extending from root, you can reach several other major system directories. By branching down from these, you can, in turn, reach all the directories and files in the file system.

In this hierarchy, files and directories that are subordinate to a directory have what is called a parent/child relationship. This type of relationship is possible for many layers of files and directories. In fact, there is no limit to the number of files and directories you may create in any directory that you own. Neither is there a limit to the number of layers of directories that you may create. Thus, you have the capability to organize your files in a variety of ways.

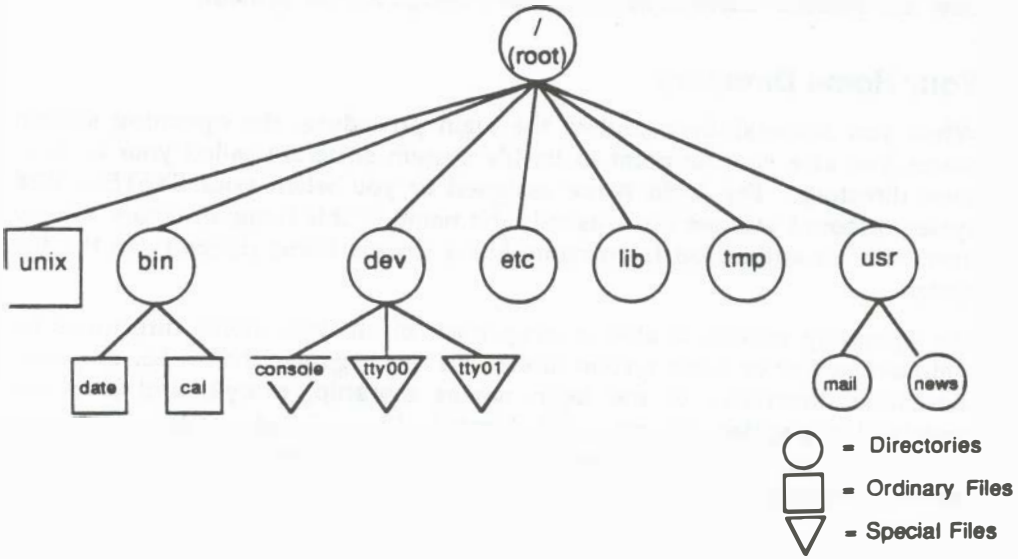


Figure 3-1. Sample File System

Your Place in the File System

Whenever you interact with the operating system, you do so from a location in its file system structure. The operating system automatically places you at a specific point in its file system every time you log in. From that point, you can move through the hierarchy to work in any of your directories and files and to access those belonging to others that you have permission to use.

The following sections describe your position in the file system structure and how this position changes as you move through the file system.

Your Home Directory

When you successfully complete the login procedure, the operating system places you at a specific point in its file system structure called your *login* or *home* directory. The login name assigned to you when your SYSTEM V/88 system account was set up is usually the name of this home directory. Every user with an authorized login name has a unique home directory in the file system.

The operating system is able to keep track of all these home directories by maintaining one or more system directories that organize them. For example, the home directories of the login names **starship**, **mary2**, and **jmrs** are contained in a system directory called **user1**. Figure 3-2 shows the position of a system directory such as **user1** in relation to the other important operating system directories discussed in Chapter 1.

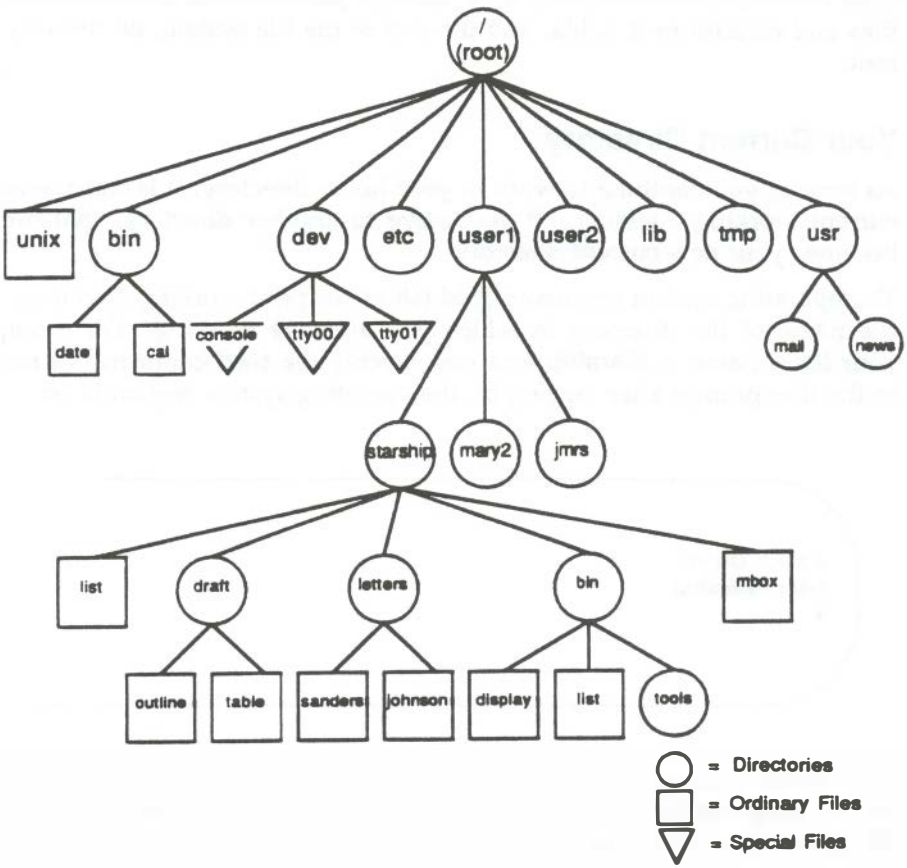


Figure 3-2. Directory of Home Directories

Within your home directory, you can create files and additional directories (sometimes called subdirectories) in which to group them. You can move and delete your files and directories, and control access to them. You have full responsibility for everything you create in your home directory because you own it. Your home directory is a vantage point from which to view all the files and directories it holds, and the rest of the file system, all the way up to root.

Your Current Directory

As long as you continue to work in your home directory, it is considered your current working directory. If you move to another directory, that directory becomes your new current directory.

The operating system command **pwd** (short for print working directory) prints the name of the directory in which you are now working. For example, if your login name is **starship** and you execute the **pwd** command in response to the first prompt after logging in, the operating system responds as:

```
$ pwd<CR>P
/user1/starship
$
```

The system response gives you both the name of the directory in which you are working (**starship**) and the location of that directory in the file system. The path name **/user1/starship** tells you that the **root** directory (shown by the leading / in the line) contains the directory **user1**, which contains the directory **starship**.

NOTE

All other slashes in the path name other than root are used to separate the names of directories and files, and to show the position of each directory relative to root.

A directory name that shows the directory's location in this way is called a full or complete directory name or path name. In the next few pages, this path name is analyzed and traced so you can start to move around in the file system.

Remember, you can determine your position in the file system at any time simply by issuing a **pwd** command. This is helpful if you want to read or copy a file and the operating system tells you the file you are trying to access does not exist. You may be surprised to find you are in a different directory than you thought.

Table 3-1 provides a summary of the syntax and capabilities of the **pwd** command.

Table 3-1. Summary of the **pwd** Command

COMMAND RECAP **pwd** – print full name of working directory		
Command	**Options**	**Arguments**
pwd	none	none
Description:	**pwd** prints the full path name of the directory in which you are currently working.	

Path Names

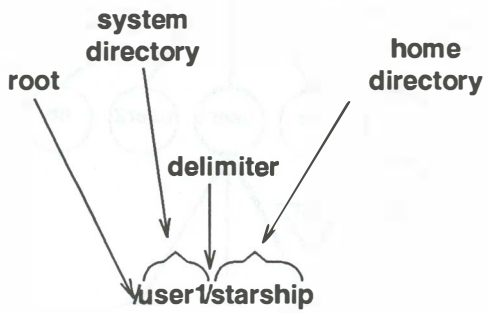
Every file and directory in the operating system is identified by a unique path name. The path name shows the location of the file or directory and provides directions for reaching it. Knowing how to follow a path name is your key to moving around the file system successfully. The first step in learning about its directions is to learn about the two types of path names: *full* and *relative.*

Full Path Names

A full path name (sometimes called an absolute path name) gives directions that start in the root directory and lead down through a unique sequence of directories to a particular directory or file. You can use a full path name to reach any file or directory.

Because a full path name always starts at the root of the file system, its leading character is always a "/" (slash). The final name in a full path name can be either a file name or a directory name. All other names in the path must be directories.

To understand how a full path name is constructed and how it directs you, consider the following example. Suppose you are working in the **starship** directory, located in **/user1**. You issue the **pwd** command and the system responds by printing the full path name of your working directory: **/user1/starship**. Analyze the elements of this path name using the following diagram and key.



/ (leading)	=	the slash that appears as the first character in the path name is the root of the file system
user1	=	system directory one level below root in the hierarchy to which root points or branches
/ (subsequent)	=	the next slash separates or delimits the directory names **user1** and **starship**
starship	=	current working directory

Now follow the bold lines in Figure 3-3 to trace the full path to **/user1/starship**.

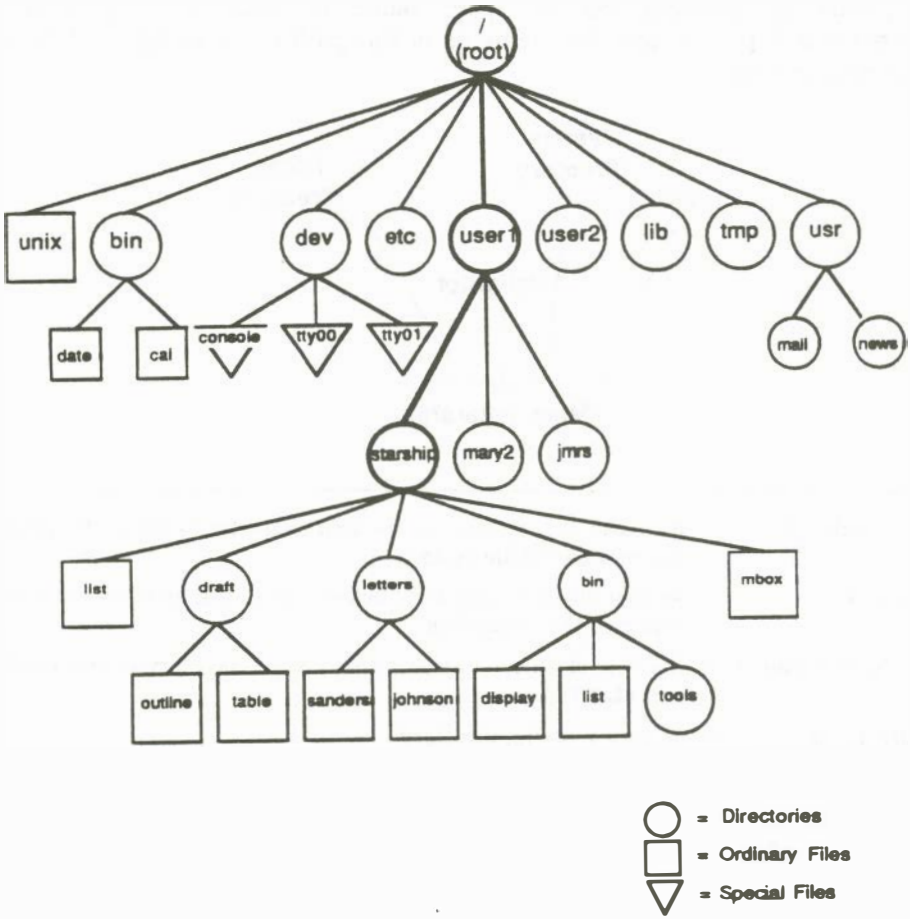


Figure 3-3. Full Path Name of the **/user1/starship** Directory

Relative Path Names

A relative path name gives directions that start in your current working directory, and lead you up or down through a series of directories to a particular file or directory. By moving down from your current directory, you can access files and directories you own. By moving up from your current directory, you pass through layers of parent directories to the grandparent of all system directories, root. From there you can move anywhere in the file system.

A relative path name begins with a directory or file name; a "." (pronounced *dot*), which is a shorthand notation for your current directory; or a ".." (pronounced *dot dot*), which is a shorthand notation for the directory immediately above your current directory in the file system hierarchy. The directory represented by ".." is called the parent directory of "." (your current directory).

For example, assume you are in the directory **starship** in the sample system and **starship** contains directories named **draft**, **letters**, and **bin** and a file named **mbox**. The relative path name to any of these is simply its name, such as **draft** or **mbox**. Figure 3-4 traces the relative path from **starship** to **draft**.

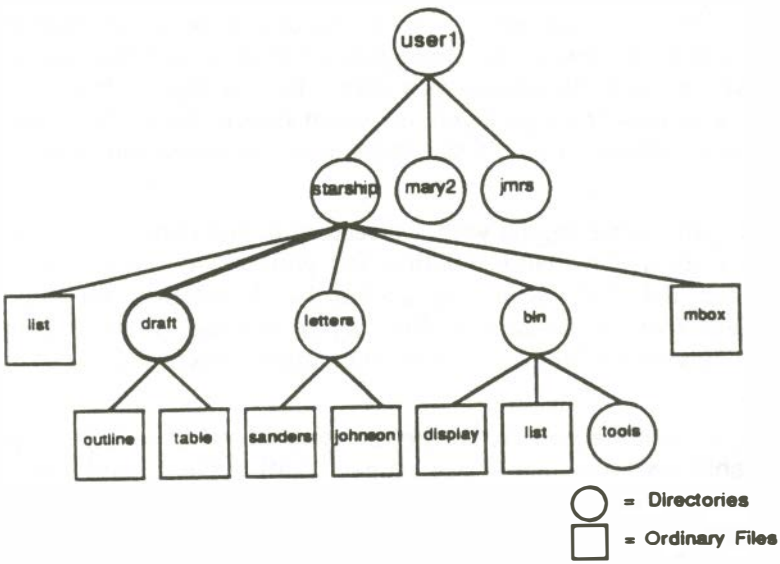


Figure 3-4. Relative Path Name of the **draft** Directory

The **draft** directory belonging to **starship** contains the files **outline** and **table**. The relative path name from **starship** to the file **outline** is **draft/outline**.

Figure 3-5 traces this relative path. Notice that the slash in this path name separates the directory named **draft** from the file named **outline**. Here, the slash is a delimiter showing that **outline** is subordinate to **draft**; i.e., **outline** is a child of its parent, **draft**.

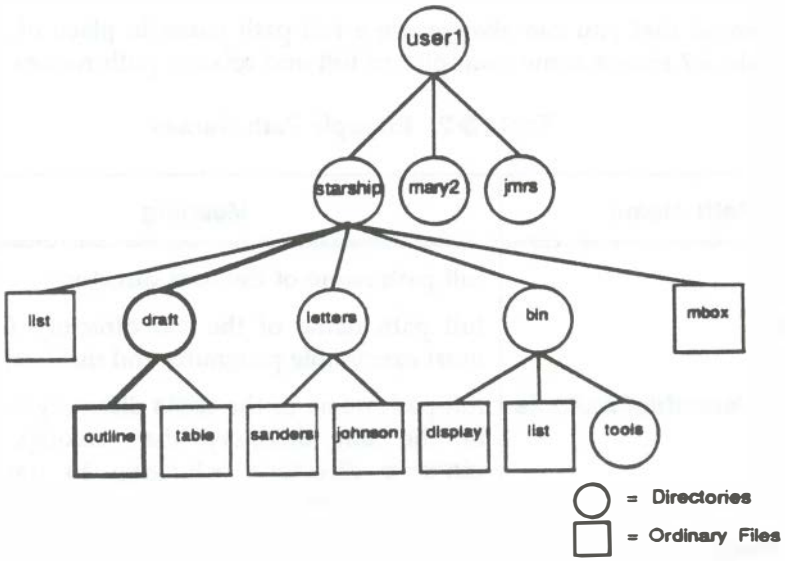


Figure 3-5. Relative Path Name from **starship** to **outline**

So far, the discussion of relative path names has covered how to specify names of files and directories that belong to, or are children of, your current directory. You now know how to move down the system hierarchy level by level until you reach your destination. You can also, however, ascend the levels in the system structure or ascend and subsequently descend into other files and directories.

To ascend to the parent of your current directory, you can use the "**..**" notation. This means that if you are in the directory named **draft** in the sample file system, "**..**" is the path name to **starship**, and "**../..**" is the path name to **starship**'s parent directory, **user1**.

From **draft**, you can also trace a path to the file **sanders** by using the path name **../letters/sanders**. The "**..**" brings you up to **starship**. Then the names **letters** and **sanders** take you down through the **letters** directory to the **sanders** file.

Keep in mind that you can always use a full path name in place of a relative one. Table 3-2 shows some examples of full and relative path names.

Table 3-2. Example Path Names

Path Name	Meaning
/	full path name of the **root** directory.
/bin	full path name of the **bin** directory (contains most executable programs and utilities).
/user1/starship/bin/tools	full path name of the **tools** directory belonging to the **bin** directory that belongs to the **starship** directory belonging to **user1** that belongs to root.
bin/tools	relative path name to the file or directory **tools** in the directory **bin**. If the current directory is **/**, then the operating system searches for **/bin/tools**. However, if the current directory is **starship**, then the system searches the full path **/user1/starship/bin/tools**.
tools	relative path name of a file or directory **tools** in the current directory.

You may need some practice before you can use path names such as these to move around the file system with confidence. However, this is to be expected when learning a new concept.

Naming Directories and Files

You can give your directories and files any names you want, as long as you observe the following rules:

- The name of a directory (or file) can be from 1 to 14 characters.
- All characters other than "/" are legal.
- Some characters are best avoided, e.g., space, tab, backspace, and the following:

  ? @ # $ ^ & * ( ) ` [ ] \ | ; ' " < >

  If you use a blank or tab in a directory or file name, you must enclose the name in quotation marks on the command line.
- Avoid using a "+", "–" or "." as the first character in a file name.
- Uppercase and lowercase characters are distinct to the operating system. For example, the system considers a directory (or file) named **draft** to be different from one named **DRAFT**.

The following are examples of legal directory or file names:

memo	**MEMO**	**section2**	**ref:list**
file.d	**chap3+4**	**item1-10**	**outline**

The rest of this chapter introduces operating system commands that enable you to examine the file system.

Organizing a Directory

This section introduces four operating system commands that enable you to organize and use a directory structure: **mkdir**, **ls**, **cd**, and **rmdir**.

mkdir
enables you to make new directories and subdirectories within your current directory.

ls
lists the names of all the subdirectories and files in a directory.

cd
enables you to change your location in the file system from one directory to another.

rmdir
enables you to remove an empty directory.

You can use these commands with either full or relative path names. Two of the commands, **ls** and **cd**, can also be used without a path name. Each command is described more fully in the following sections.

Creating Directories: mkdir Command

It is recommended that you create subdirectories in your home directory according to a logical and meaningful scheme that will facilitate the retrieval of information from your files. If you put all files about one subject in a directory, you will know where to find them later.

To create a directory, use the command **mkdir** (short for *make directory*). Enter the command name, followed by the name you are giving your new directory. For example, in the sample file system, the owner of the **draft** subdirectory created **draft** by issuing the following command from the home directory (**/user1/starship**):

```
$ mkdir draft <CR>
$
```

The second prompt shows that the command has succeeded; the subdirectory **draft** has been created.

Still in the home directory, this user created other subdirectories, e.g., **letters** and **bin**, in the same way:

```
$ mkdir letters<CR>
$ mkdir bin<CR>
$
```

The user could have created all three subdirectories (**draft**, **letters**, and **bin**) simultaneously by listing them all on a single command line:

```
$ mkdir draft letters bin<CR>
$
```

You can also move to a subdirectory you created and build additional subdirectories within it. When you build directories or create files, you can name them anything you want as long as you follow the guidelines listed earlier under *Naming Directories and Files.*

Table 3-3 summarizes the syntax and capabilities of the **mkdir** command.

Table 3-3. Summary of the **mkdir** Command

Command Recap **mkdir** – make a new directory		
Command	**Options**	**Arguments**
mkdir	available*	*directoryname(s)*
Description:	**mkdir** creates a new directory (subdirectory).	
Remarks:	The system returns a prompt (**$** by default) if the directory is successfully created.	

* See the **mkdir** page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Listing Directory Contents: ls Command

All directories in the file system have information about the files and directories they contain, e.g., name, size, and the date last modified. You can obtain this information about the contents of your current directory and other system directories by executing the command **ls** (short for list).

The **ls** command lists the names of all files and subdirectories in a specified directory. If you do not specify a directory, **ls** lists the names of files and directories in your current directory. To understand how the **ls** command works, consider the sample file system (Figure 3-2) once again:

> You are logged in to the operating system and you run the **pwd** command. The system responds with the path name **/user1/starship**. To display the names of files and directories in this current directory, you then type **ls** and press the RETURN key. After this sequence, your terminal will read:

```
$ pwd<CR>
$/user1/starship
$ ls<CR>
bin
draft
letters
list
mbox
$
```

The system responds by listing, in alphabetical order, the names of files and directories in the current directory **starship**. (If the first character of any of the file or directory names had been a number or an uppercase letter, it would print first.)

To print the names of files and subdirectories in a directory other than your current directory without moving from your current directory, you specify the name of that directory as:

ls *directoryname*<**CR**>

The directory name can be either the full or relative path name of the desired directory. For example, you can list the contents of **draft** while you are working in **starship** by entering **ls draft** and pressing the RETURN key. Your screen displays:

```
$ ls draft<CR>
outline
table
$
```

Here, **draft** is a relative path name from a parent (**starship**) to a child (**draft**) directory.

You can also use a relative path name to print the contents of a parent directory when you are located in a child directory. The ".." (dot dot) notation provides an easy way to do this. For example, the following command line specifies the relative path name from **starship** to **user1**:

```
$ ls ..<CR>
jmrs
mary2
starship
$
```

You can get the same results by using the full path name from root to **user1**.

If you type **ls /user1** and press the **RETURN** key, the system responds by printing the same list. You can also list the contents of any system directory that you have permission to access by executing the **ls** command with a full or relative path name.

The **ls** command is useful if you have a long list of files and you are trying to determine if one of them exists in your current directory. For example, if you are in the directory **draft** and you want to determine if the files named **outline** and **notes** are there, use the **ls** command as follows:

```
$ ls outline notes<CR>
outline
notes not found
$
```

The system acknowledges the existence of **outline** by printing its name, and says that the file **notes** is not found.

The **ls** command does not print the contents of a file. If you want to see what a file contains, use the **cat**, **pg**, or **pr** command; they are described in *Accessing and Manipulating Files*, later in this chapter.

Frequently Used ls Options

The **ls** command also accepts options that cause specific attributes of a file or subdirectory to be listed. There are more than a dozen available options for the **ls** commands. Of these, the **−a** and **−l** will probably be most valuable in your basic use of the operating system. Refer to the **ls**(1) page in the *User's Reference Manual* for details about other options.

Listing All Names in a File

Some important file names in your home directory, e.g., **.profile** (pronounced *dot-profile*), begin with a period. When a file name begins with a dot, it is not included in the list of files reported by the **ls** command. If you want the **ls** to include these files, use the **–a** option on the command line. For example, to list all the files in your current directory (**starship**), including those that begin with a "."(dot), type **ls –a** and press the **RETURN** key:

```
$ ls -a<CR>
.
..
.profile
bin
draft
letters
list
mbox
$
```

Listing Contents in Short Format

The **–C** and **–F** options for the **ls** command are frequently used. Together, these options list a directory's subdirectories and files in columns, and identify executable files (with an * (prints * centered on line) and directories (with a "/"). Thus, you can list all files in your working directory **starship** by executing the command line shown here:

```
$ ls -CF<CR>[
bin/    letters/        mbox
draft/  list*
$[
```

Listing Contents in Long Format

Probably the most informative **ls** option is **–l**, which displays the contents of a directory in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file. For example, run the **ls –l** command while in the **starship** directory.

```
$ ls -l<CR>
total 30
drwxr-xr-x   3 starship   project      96  Oct 27  08:16  bin
drwxr-xr-x   2 starship   project      64  Nov  1  14:19  draft
drwxr-xr-x   2 starship   project      80  Nov  8  08:41  letters
-rwx------   2 starship   project   12301  Nov  2  10:15  list
-rw-------   1 starship   project      40  Oct 27  10:00  mbox
$
```

The first line of output (`total 30`) shows the amount of disk space used, measured in blocks. Each of the rest of the lines comprises a report on a directory or file in **starship**. The first character in each line (`d`, `-`, `b`, or `c`) tells you the type of file.

`d` = directory

`-` = ordinary disk file

`b` = block special file

`c` = character special file

Using this key to interpret the previous screen, you can see that the **starship** directory contains three directories and two ordinary disk files.

The next several characters, which are either letters or hyphens, identify who has permission to read and use the file or directory. (Permissions are discussed in the description of the **chmod** command under *Accessing and Manipulating Files* later in this chapter.)

The following number is the link count. For a file, this equals the number of other files linked to that file. For a directory, this number shows the number of directories immediately under it plus two (for the directory itself and its parent directory).

Next, the login name of the file's owner appears (here it is **starship**), followed by the group name of the file or directory (**project**).

The following number shows the length of the file or directory entry measured in units of information (or memory) called bytes. The month, day, and time that the file was last modified are given next. Finally, the last column shows the name of the directory or file.

Figure 3-6 identifies each column in the rows of output from the **ls –l** command.

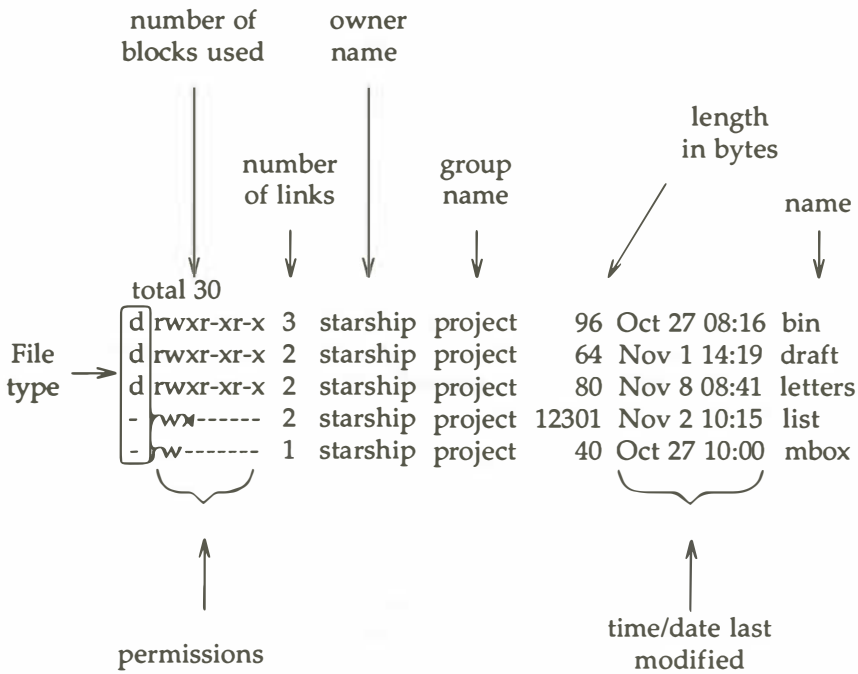


Figure 3-6. Description of Output Produced by the **ls –l** Command

Table 3-4 summarizes the syntax and capabilities of the **ls** command and two available options.

Table 3-4. Summary of the **ls** Command

Command Recap **ls** – list contents of a directory		
Command	**Options**	**Arguments**
ls	**–a, –l,** and others*	*directoryname(s)* or *filename(s)*
Description:	**ls** lists the names of the files and subdirectories in the specified directories. If no directory name is given as an argument, lists the contents of your working directory.	
Options:	**–a** Lists all entries, including those beginning with "." (dot). **–l** Lists contents of a directory in long format, furnishing mode, permissions, size in bytes, and time of last modification.	
Remarks:	If you want to read the contents of a file, use the **cat** command.	

* See the **ls**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Changing Your Current Directory: cd Command

When you first log in on the operating system, you are placed in your home directory. As long as you do work in it, it is also your current working directory. However, by using the command **cd** (short for change directory), you can also work in other directories. To use this command, enter **cd**, followed by a path name to the directory to which you want to move:

cd *pathname_of_newdirectory*<**CR**>

Any valid path name (full or relative) can be used as an argument to the **cd** command. If you do not specify a path name, the command moves you to your home directory. Once you have moved to a new directory, it becomes your current directory.

For example, to move from the **starship** directory to its child directory **draft** (in the sample file system), type **cd draft** and press the RETURN key. (Here **draft** is the relative path name to the desired directory.) When you get a prompt, verify your new location by typing **pwd** and pressing the RETURN key. Your terminal screen appears as:

```
$ cd draft<CR>
$ pwd<CR>
/user1/starship/draft
$
```

Now that you are in the **draft** directory you can create subdirectories in it by using the **mkdir** command, and new files, by using the **ed** and **vi** editors. (Refer to Chapters 5 and 6 for tutorials on the **ed** and **vi** commands, respectively.)

It is not necessary to be in the **draft** directory to access files within it. You can access a file in any directory by specifying a full path name for it. For example, to display the **sanders** file in the **letters** directory (**/user1/starship/letters**) while you are in the **draft** directory, **/user1/starship/draft**, specify the full path name of **sanders** on the command line when you issue **cat** command (more on the **cat** command later):

cat /user1/starship/letters/sanders<CR>

You may also use full path names with the **cd** command. For example, to move to the **letters** directory from the **draft** directory, specify **/user1/starship/letters** on the command line:

cd /user1/starship/letters<CR>

Also, because **letters** and **draft** are both children of **starship**, you can use the relative path name **../letters** with the **cd** command. The "**..**" notation moves you to the directory **starship**, and the rest of the path name moves you to **letters**.

Table 3-5 summarizes the syntax and capabilities of the **cd** command.

Table 3-5. Summary of the **cd** Command

COMMAND RECAP **cd** – change your working directory		
Command	**Options**	**Arguments**
cd	none	*directoryname*
Description:	**cd** changes your position in the file system from the current directory to the directory specified. If no directory name is given as an argument, the **cd** command places you in your home directory.	
Remarks:	When the shell places you in the directory specified, the prompt (**$** by default) is returned to you. To access a directory that is not in your working directory, you must use the full or relative path name in place of a simple directory name.	

Removing Directories: rmdir Command

If you no longer need a directory, you can remove it with the command **rmdir** (short for remove a directory). The standard syntax for this command is:

rmdir *directoryname(s)*<**CR**>

You can specify more than one directory name on the command line.

The **rmdir** command will not remove a directory if you are not the owner of it or if the directory is not empty. If you want to remove a file in another user's directory, the owner must give you write permission for the parent directory of the file you want to remove.

If you try to remove a directory that still contains subdirectories and files (i.e., it is not empty), the **rmdir** command prints the message *directoryname* **not empty**. You must remove all subdirectories and files (including files whose filename starts with "."); only then will the command succeed.

For example, you have a directory called **memos** that contains one subdirectory, **tech**, and two files, **june.30** and **july.31**. (Create this directory in your home directory now so you can see how the **rmdir** command works.) If you try to remove the directory **memos** (by issuing the **rmdir** command from your home directory), the command responds as:

```
$ rmdir memos<CR>
rmdir:  memos not empty
$
```

To remove the directory **memos**, you must first remove its contents: the subdirectory **tech**, and the files **june.30** and **july.31**. You can remove the **tech** subdirectory by executing the **rmdir** command. For instructions on removing files, see *Accessing and Manipulating Files* later in this chapter.

Once you have removed the contents of the **memos** directory, **memos** can be removed. First, however, you must move to its parent directory (your home directory). The **rmdir** command will not work if you are still in the directory you want to remove. From your home directory, type:

rmdir memos<CR>

If **memos** is empty, the command removes it and returns a prompt.

Table 3-6 summarizes the syntax and capabilities of the **rmdir** command.

Table 3-6. Summary of the **rmdir** Command

Command Recap **rmdir** – remove a directory		
Command	**Options**	**Arguments**
rmdir	available*	*directoryname(s)*
Description:	**rmdir** removes specified directories only when they do not contain files or subdirectories.	
Remarks:	If the directory is empty, it is removed and the system returns a prompt. When the directory contains files or subdirectories, the command returns the message, `rmdir:` *directoryname* `not empty`.	

* See the **rmdir** page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Accessing and Manipulating Files

This section presents several operating system commands that access and manipulate files in the file system structure. Information in this section is organized into two parts: basic and advanced. The part devoted to basic commands is fundamental to using the file system; the advanced commands offer more sophisticated information processing techniques for working with files.

Basic Commands

This section discusses SYSTEM V/88 commands that are necessary for accessing and using the files in the directory structure. Table 3-7 lists these commands.

Table 3-7. Basic Commands for Using Files

Command	Function
cat	displays the contents of a specified file on a terminal.
pg	displays the contents of a specified file on a terminal in chunks or pages.
pr	displays a partially formatted version of a specified file on the terminal.
lp	requests a paper copy of a file from a line printer.
cp	makes a duplicate copy of an existing ordinary file.
mv	moves and renames an ordinary or directory file.
rm	removes an ordinary or special file.
wc	reports the number of lines, words, and characters in an ordinary file.
chmod	changes permission modes for a file (or a directory).

Each command is discussed in detail and summarized for easy reference later. These summaries will allow you to review the syntax and capabilities of these commands at a glance.

Displaying a File's Contents: cat, pg, and pr

The operating system has three commands to display and print the contents of a file or files: **cat, pg,** and **pr**. The **cat** command (short for concatenate) outputs the contents of the files specified. This output displays on your screen unless you tell **cat** to direct it to another file or a new command.

The **pg** command is useful when you want to read the contents of a long file because it displays the text of a file in pages a screenful at a time.

The **pr** command formats specified files and displays them on your screen or, if you so request, directs the formatted output to a printer (refer to the **lp** command in this chapter).

The following sections describe how to use the **cat, pg,** and **pr** commands.

Concatenate and Print Contents of a File: the cat Command

The **cat** command displays the contents of an ordinary file or files. For example, you are located in the directory **letters** (in the sample file system) and you want to display the contents of the file **johnson**. Type the command line shown on the screen and you will receive the following output:

```
$ catjohnson<CR>
March 5, 1986

Mr. Ron Johnson
Layton Printing
52 Hudson Street
New York, N.Y.

Dear Mr. Johnson:

I enjoyed speaking with you this morning
about your company's plans to automate
your business.
Enclosed please find
the material you requested
about AB&C's line of computers
and office automation software.

If I can be of further assistance to you,
please don't hesitate to call.

Yours truly,

John Howe
$
```

To display the contents of two (or more) files, type the names of the files you want to see on the command line. For example, to display the contents of the files **johnson** and **sanders**:

$ **cat johnson sanders<CR>**

The **cat** command reads **johnson** and **sanders** and displays their contents in that order on your screen:

```
$ cat johnson sanders<CR>
March 5, 1986

Mr. Ron Johnson
Layton Printing
52 Hudson Street
New York, N.Y.

Dear Mr. Johnson:

I enjoyed speaking with you this morning
.
.

Yours truly,

John Howe


March 5, 1986

Mrs. D.L. Sanders
Sanders Research, Inc.
43 Nassau Street
Princeton, N.J.

Dear Mrs. Sanders:

My colleagues and I have been following, with great interest,
.
.

Sincerely,

John Howe
$
```

To direct the output of the **cat** command to another file or to a new command, refer to the sections in Chapter 7 that discuss input and output redirection. Table 3-8 summarizes the syntax and capabilities of the **cat** command.

Table 3-8. Summary of the **cat** Command

Command Recap **cat** – concatenate and print a file's contents		
Command	**Options**	**Arguments**
cat	available*	*filelist*
Description:	The **cat** command reads the name of each file specified on the command line and displays its contents.	
Remarks:	If a specified file exists and is readable, its contents display on the screen; otherwise, the message `cat: cannot open` *filename* appears on the screen. To display the contents of a directory, use the **ls** command.	

* See the **cat**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

NOTE: To prevent terminal settings from being changed, only ordinary files that are printable should be used by **cat** (i.e., not executable or data files).

Paging Through the Contents of a File: pg Command

The command **pg** (short for page) allows you to examine the contents of files page by page, on a terminal. The **pg** command displays the text of a file in pages (chunks) followed by a colon prompt (:), a signal that the program is waiting for your instructions. Possible instructions you can then issue include requests for **pg** to continue displaying the file's contents a page at a time, and a request that **pg** search through the files to locate a specific character pattern. Table 3-9 summarizes some of the available instructions.

Table 3-9. Summary of Commands to Use with **pg**

pg Command*	Function
h	displays list of available **pg**† commands.
q or **Q**	quits **pg** perusal mode.
<**CR**>	displays next page of text.
l	displays next line of text.
d or ^**d**	displays additional half page of text.
. or ^**l**	redisplays current page of text.
f	skips next page of text and displays the following one.
n	begins displaying next file you specified on command line.
p	displays previous file specified on command line.
$	displays last page of text in file currently displayed.
/pattern	searches forward in file for specified character pattern.
?*pattern*	searches backward in file for specified character pattern.
!	escapes from file to the shell.

* Most commands can be typed with a number preceding them. For example, +1 (display next page), −1 (display previous page), or 1 (display first page of text).

† See the *User's Reference Manual* for an explanation of all available **pg** commands.

The **pg** command is useful when you want to read a long file or a series of files because the program pauses after displaying each page, allowing time to examine it. The size of the page displayed depends on the terminal. For example, on a terminal capable of displaying 24 lines, one page is defined as 23 lines of text and a line containing a colon. However, if a file is less than twenty-three lines long, its page size is the number of lines in the file plus one (for the colon).

To read the contents of a file with **pg**, use the following command line format:

pg *filename(s)*<**CR**>

For example, to display the contents of the file **outline** in the sample file system, type:

pg outline<**CR**>

The first page of the file appears on the screen. Because the file has more lines in it than can display on one page, a colon appears at the bottom of the screen. This is a reminder to you that there is more of the file to be seen. When you are ready to read more, press the **RETURN** key and **pg** prints the next page of the file.

The following screen depicts our discussion of the **pg** command:

```
$ pg outline<CR>
After you analyze the subject for your
report, you must consider organizing and
arranging the material you want to use in
writing it.
      .
      .
      .
An outline is an effective method of
organizing the material.  The outline
is a type of blueprint or skeleton,
a framework for you the builder-writer
of the report; in a sense it is a recipe
:<CR>
```

After you press the **RETURN** key, **pg** resumes printing the file's contents on the screen.

```
that contains the names of the
ingredients and the order in which
to use them.
      .
      .
      .
Your outline need not be elaborate or
overly detailed; it is simply a guide you
may consult as you write, to be varied,
if need be, when additional important
ideas are suggested in the actual writing.
(EOF):
```

Notice the line at the bottom of the screen that contains the string `(EOF):`. This expression `(EOF)` means you have reached the end of the file. The colon prompt is a cue for you to issue another command.

When you have finished examining the file, press the **RETURN** key; a prompt appears on your terminal. (Typing **q** or **Q** and pressing the **RETURN** key also gives you a prompt.) Or you can use one of the other available commands, depending on your needs. In addition, there are a number of options that can be specified on the **pg** command line. Refer to the **pg**(1) page in the *User's Reference Manual*.

Proper execution of the **pg** command depends on specifying the type of terminal you are using. This is because the **pg** program was designed to be flexible enough to run on many different terminals; how it is executed differs from terminal to terminal. By specifying one type, you are telling this command:

- how many lines to print
- how many columns to print
- how to clear the screen
- how to highlight prompt signs or other words
- how to erase the current line

To specify a terminal type, assign the code for your terminal to the **TERM** variable in your **.profile** file. (For more information about **TERM** and **.profile**, see Chapter 7; for instructions on setting the **TERM** variable, see Appendix F.)

Table 3-10 summarizes the syntax and capabilities of the **pg** command.

Table 3-10. Summary of the **pg** Command

Command Recap **pg** – display a file's contents in chunks or pages		
Command	**Options**	**Arguments**
pg	available*	*filelist*
Description:	The **pg** command displays the contents of the specified files in pages.	
Remarks:	After displaying a page of text, the **pg** command awaits instructions from you to do one of the following: continue to display text, search for a pattern of characters, or exit the **pg** view mode. In addition, a number of options are available. For example, you can display a section of a file beginning at a specific line or at a line containing a certain sequence or pattern. You can also opt to go back and review text that has already displayed.	

* See the **pg**(1) page in the *User's Reference Manual* for all options and an explanation of their capabilities.

Making a Copy of a File: cp Command

When using the operating system, you may want to make a copy of an ordinary file. For example, revise a file while leaving the original version intact. The command **cp** (short for copy) copies the complete contents of one file into another. The **cp** command also allows you to copy one or more files from one directory into another while leaving the original file or files in place.

To copy the file named **outline** to a file named **new.outline** in the sample directory, type **cp outline new.outline** and press the RETURN key. The system returns the prompt when the copy is made. To verify the existence of the new file, you can type **ls** and press the RETURN key. This command lists the names of all files and directories in the current directory, in this case **draft**. The following screen depicts these commands:

```
$ cp outline new.outline<CR>
$ ls<CR>
new.outline
outline
table
$
```

The operating system does not allow you to have two files with the same name in a directory. In this case, because there was no file called **new.outline** when the **cp** command was issued, the system created a new file with that name. However, if a file called **new.outline** had already existed, it would have been replaced by a copy of the file **outline**; the previous version of **new.outline** would have been deleted.

If you had tried to copy the file **outline** to another file named **outline** in the same directory, the system would have told you the file names were identical and returned the prompt to you.

If you had then listed the contents of the directory to determine exactly how many copies of **outline** existed, you would have received the following on your screen:

```
$ cp outline outline<CR>
cp: outline and outline are identical
$ ls<CR>
outline
table
$
```

The operating system does allow you to have two files with the same name as long as they are in different directories. For example, the system would let you copy the file **outline** from the **draft** directory to another file named **outline** in the **letters** directory.

If you were in the **draft** directory, you could use any one of four command lines. In the first two command lines, you specify the name of the new file you are creating by making a copy:

cp outline /user1/starship/letters/outline<CR> (full path name specified)

cp outline ../letters/outline<CR> (relative path name specified)

NOTE

If the directory that we assumed didn't really exist, then the file copy would instead receive the name of the innermost directory.

However, the **cp** command does not require that you specify the name of the new file. If you do not include a name for it on the command line, **cp** gives your new file the same name as the original one, by default. Therefore you could also use either of these command lines:

cp outline /user1/starship/letters<CR> (full path name specified)

cp outline ../letters<CR> (relative path name specified)

In all four cases, **cp** makes a copy of the **outline** file in the **letters** directory and calls it **outline**, too. If you want to give your new file a different name, you must specify it. For example, to copy the file **outline** in the **draft** directory to a file named **outline.vers2** in the **letters** directory, you can use either of these commands:

cp outline /user1/starship/letters/outline.vers2<CR> (full path name)

cp outline ../letters/outline.vers2<CR> (relative path name)

When assigning new names, remember the rules described in *Naming Directories and Files* in this chapter.

Table 3-11 summarizes the syntax and capabilities of the **cp** command.

Table 3-11. Summary of the **cp** Command

Command Recap **cp** – make a copy of a file		
Command	**Options**	**Arguments**
cp	none	*file1 file2* *file(s) directory*
Description:	**cp** allows you to make a copy of *file1* and call it *file2* leaving *file1* intact or to copy one or more files into a different directory.	
Remarks:	When you are copying *file1* to *file2* and a file called *file2* already exists, the **cp** command overwrites the first version of *file2* with a copy of *file1* and calls it *file2*. The first version of *file2* is deleted. You cannot copy directories or special files with the **cp** command.	

Moving and Renaming a File: mv Command

The command **mv** (short for move) allows you to rename a file in the same directory or to move a file from one directory to another. If you move a file to a different directory, the file can be renamed or it can retain its original name.

To rename a file within one directory, follow this format:

mv *file1 file2*<**CR**>

The **mv** command changes a file's name from *file1* to *file2* and deletes *file1*. Remember that the names *file1* and *file2* can be any valid names, including path names.

For example, if you are in the directory **draft** in the sample file system and you would like to rename the file **table** to **new.table**, type **mv table new.table** and press the **RETURN** key. If the command executes successfully, you receive a prompt. To verify that the file **new.table** exists, you can list the contents of the directory by typing **ls** and pressing the **RETURN** key. The screen shows your input and the system's output:

```
$ mv table new.table<CR>
$ ls<CR>
new.table
outline
$
```

You can also move a file from one directory to another, keeping the same name or changing it to a different one. To move the file without changing its name, use the following command line:

mv *file(s) directory*<**CR**>

The file and directory names can be any valid names, including path names.

For example, you want to move the file **table** from the current directory named **draft** (whose full path name is **/user1/starship/draft**) to a file with the same name in the directory **letters** (whose relative path name from **draft** is **../letters** and whose full path name is **/user1/starship/letters**), you can use any one of several command lines:

mv table /user1/starship/letters<CR>

mv table /user1/starship/letters/table<CR>

mv table ../letters<CR>

mv table ../letters/table<CR>

mv /user1/starship/draft/table /user1/starship/letters/table<CR>

If you want to rename the file **table** as **table2** when moving it to the directory **letters**, use any of these command lines:

mv table /user1/starship/letters/table2<CR>

mv table ../letters/table2<CR>

mv /user1/starship/draft/table2 /user1/starship/letters/table2<CR>

You can verify that the command worked by using the **ls** command to list the contents of the directory.

Table 3-12 summarizes the syntax and capabilities of the **mv** command.

Table 3-12. Summary of the **mv** Command

Command Recap **mv** – move or rename files		
Command	**Options**	**Arguments**
mv	available*	*file1 file2* *file(s) directory*
Description:	**mv** allows you to change the name of a file or to move a file(s) into another directory.	
Remarks:	When you are moving *file1* to *file2*, if a file called *file2* already exists, the **mv** command overwrites the first version of *file2* with *file1* and renames it *file2*. The first version of *file2* is deleted.	

* See **mv**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Removing a File: rm Command

When you no longer need a file, you can remove it from your directory by executing the command **rm** (short for remove). The basic format for this command is:

rm *file(s)*<**CR**>

You can remove more than one file at a time by specifying those files you want to delete on the command line with a space separating each filename:

rm *file1 file2 file3*<**CR**>

The system does not save a copy of a file it removes; once you have executed this command, your file is removed permanently.

After you have issued the **rm** command, you can verify its successful execution by running the **ls** command. Because **ls** lists the files in your directory, you'll immediately be able to see whether or not **rm** has executed successfully.

For example, suppose you have a directory that contains two files, **outline** and **table**. You can remove both files by issuing the **rm** command once. If **rm** is executed successfully, your directory is empty. Verify this by running the **ls** command.

```
$ rm outline table <CR>
$ ls
$
```

The prompt shows that **outline** and **table** were removed.

Table 3-13 Summarizes the syntax and capabilities of the **rm** command.

Table 3-13. Summary of the **rm** Command

Command Recap **rm** – remove a file		
Command	**Options**	**Arguments**
rm	available*	*filelist*
Description:	**rm** allows you to remove one or more files.	
Remarks:	Files specified as arguments to the **rm** command are removed permanently.	

* See the **rm**(1) page in the *User's Reference Manual* for all options and an explanation of their capabilities.

Counting Lines, Words, and Characters in a File: wc Command

The command **wc** (short for word count) reports the number of lines, words, and characters there are in the ordinary files named on the command line. If you name more than one file, the **wc** program counts the number of lines, words, and characters in each specified file and then totals the counts. In addition, you can direct the **wc** program to give you only a line, a word, or a character count by using the **−l**, **−w**, or **−c** options, respectively.

To determine the number of lines, words, and characters in a file, use the following format on the command line:

wc *file1*<CR>

The system responds with a line in the following format:

l w c file1

where:

l represents the number of lines in *file1*.

w represents the number of words in *file1*.

c represents the number of characters in *file1*.

For example, to count the lines, words, and characters in the file **johnson** (located in the current directory, **letters**), type the following command line:

```
$ wc johnson<CR>
24      66      406 johnson
$
```

The system response means that the file **johnson** has 24 lines, 66 words, and 406 characters.

To count the lines, words, and characters in more than one file, use this format:

wc *file1 file2*<**CR**>

The system responds in the following format:

```
l    w    c    file1
l    w    c    file2
l    w    c    total
```

Line, word, and character counts for *file1* and *file2* display on separate lines and the combined counts appear on the last line beside the word `total`.

For example, ask the **wc** program to count the lines, words, and characters in the files **johnson** and **sanders** in the current directory:

```
$ wc johnson sanders<CR>
     24      66      406 johnson
     28      92      559 sanders
     52     158      965 total
$
```

The first line reports that the **johnson** file has 24 lines, 66 words, and 406 characters. The second line reports 28 lines, 92 words, and 559 characters in the **sanders** file. The last line shows that these two files together have a total of 52 lines, 158 words, and 965 characters.

To get only a line, a word, or a character count, select the appropriate command line format from the following lines:

wc **−l** *file1*<**CR**> (line count)
wc **−w** *file1*<**CR**> (word count)
wc **−c** *file1*<**CR**> (character count)

For example, using the **−l** option, the system reports only the number of lines in **sanders**:

```
$ wc -l sanders<CR>
      28 sanders
$
```

If the **-w** or **-c** option had been specified instead, the command would have reported the number of words or characters, respectively, in the file.

Table 3-14 summarizes the syntax and capabilities of the **wc** command.

Table 3-14. Summary of the **wc** Command

Command Recap **wc** – count lines, words, and characters in a file		
Command	**Options**	**Arguments**
wc	**-l, -w, -c**	*filelist*
Description:	**wc** counts lines, words, and characters in the specified files, keeping a total count of all tallies when more than one file is specified.	
Options	**-l** counts the number of lines in the specified file(s). **-w** counts the number of words in the specified file(s). **-c** counts the number of characters in the specified file(s).	
Remarks:	When a file name is specified in the command line, the counts requested are displayed.	

Protecting Your Files: chmod Command

The command **chmod** (short for change mode) allows you to decide who can use your files and who cannot. Because the operating system is a multi-user system, you usually do not work alone in the file system. System users can follow path names to various directories and read and use files belonging to one another, as long as they have permission to do so.

If you own a file, you can decide who has the right to read it, write in it (make changes to it), or, if it is a program, execute it. You can also restrict permissions for directories with the **chmod** command. When you grant execute permission for a directory, you allow the specified users to **cd** to it and list its contents with the **ls** command.

To assign or remove these types of permissions, use the following three symbols:

- **r** allows system users to read a file or to copy its contents.
- **w** allows system users to write changes into a file (or a copy of it).
- **x** allows system users to run an executable file.

To specify the users to whom you are granting or denying these types of permission, use these three symbols:

- **u** indicates you, the owner of your files and directories (where **u** is short for user).
- **g** indicates members of the group to which you belong. (The group could consist of team members working on a project, members of a department, or a group arbitrarily designated by the person who set up your operating system account.)
- **o** indicates all other system users.

When you create a file or a directory, the system automatically grants or denies permission to you, members of your group, and other system users. You can alter this automatic action by modifying your environment (see "umask" in Chapter 7 for details). Moreover, regardless of how the permissions are granted when a file is created, as the owner of the file or directory you always have the option of changing them. For example, you may want to keep certain files private and reserve them for your exclusive use. You may want to grant permission to read and write changes into a file to members of your group and all other system users as well. Or you may

share a program with members of your group by granting them permission to execute it.

Determining Existing Permissions

You can determine what permissions are currently in effect on a file or a directory by using the command that produces a long listing of a directory's contents: **ls −l**. For example, typing **ls −l** and pressing the RETURN key while in the directory named **starship/bin** in the sample file system produces the following output:

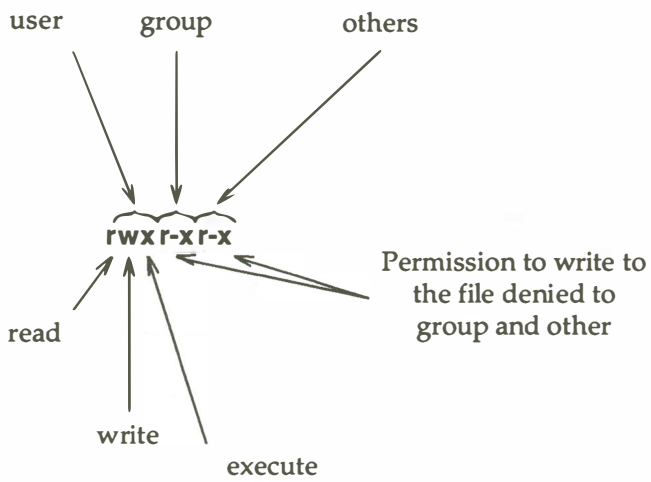
```
$ ls -l<CR>
total 35
-rwxr-xr-x  1 starship   project   9346  Nov 1  08:06  display
-rw-r--r--  1 starship   project   6428  Dec 2  10:24  list
drwx--x--x  2 starship   project     32  Nov 8  15:32  tools
$
```

Permissions for the **display** and **list** files and the **tools** directory are shown on the left of the screen under the line `total 35`, and appear in this format:

`-rwxr-xr-x` (for the **display** file)
`-rw-r--r--` (for the **list** file)
`drwx--x---` (for the **tools** directory)

After the initial character, which describes the file type (e.g., a `-` (dash) for a regular file or a `d` for a directory), there are nine characters that set the permissions. They are composed of three sets of three characters. The first set refers to permissions for the owner, the second set to permissions for group members, and the last set to permissions for all other system users. Within each set of characters, the `r`, `w`, and `x` show the permissions currently granted to each category. If a dash appears instead of an `r`, `w`, or `x`, permission to read, write, or execute is denied.

The following diagram summarizes this breakdown for the file named **display**.



As you can see, the owner has read (**r**), write (**w**), and execute (**x**) permissions. The group and other system users have read (**r**) and execute (**x**) permissions.

There are two exceptions to this notation system. Occasionally, the letter **s** or the letter **l** may appear in the permissions line, instead of an **r**, **w**, or **x**. The letter **s** (short for set user ID or set group ID) represents a special type of permission to execute a file. It appears where you normally see an **x** (or –) for the user or group (the first and second sets of permissions). From a user's point of view, it is equivalent to an **x** in the same position; it implies that execute permission exists. It is significant only for programmers and system administrators. (See the *System Administrator's Guide* for details about setting the user or group ID.)

The letter **l** is the symbol for lock enabling. It does not mean that the file has been locked. It simply means that the function of locking is enabled, or possible, for this file. The file may or may not be locked; that cannot be determined by the presence or absence of the letter **l**.

The letter **t** means that the file has the "sticky" attribute assigned.

Changing Existing Permissions

After you have determined what permissions are in effect, you can change them by executing the **chmod** command in the following format:

chmod *who+permission filelist*<**CR**>

or

chmod *who−permission filelist*<**CR**>

The following list defines each component of this command line:

CHMOD	Name of the program
who	one of three user groups: **u** = user **g** = group **o** = others
+ or −	instruction that grants (+) or denies (−) permission
permission	any combination of three authorizations: **r** = read **w** = write **x** = execute
filelist	file (or directory) name(s) listed; assumed to be branches from your current directory, unless you use full pathnames.

NOTE

The **chmod** command will not work if you type any spaces between *who*, the instruction that gives (+) or denies (−) permission, and the *permission*.

The following examples show a few ways to use the **chmod** command. As the owner of **display**, you can read, write, and run this executable file. You can protect the file against being accidentally changed by denying yourself write (**w**) permission. To do this, type the command line:

chmod u−w display<CR>

After receiving the prompt, type **ls −l** and press the RETURN key to verify that this permission has been changed, as shown in the following screen:

```
$ chmod u-w display<CR>
$ ls -l<CR>
total 35
-r-xr-xr-x  1 starship   project   9346  Nov 1  08:06  display
rw-r--r--   1 starship   project   6428  Dec 2  10:24  list
drwx--x--x  2 starship   project     32  Nov 8  15:32  tools
$
```

You no longer have permission to write changes into the file. You will not be able to change this file until you restore write permission for yourself.

Now consider another example. Notice that permission to write into the file **display** has been denied to members of your group and other system users. However, they do have read permission. This means they can copy the file into their own directories and then make changes to it. To prevent all system users from copying this file, you can deny them read permission by typing:

chmod go−r display<CR>

The **g** and **o** stand for group members and all other system users, respectively. The **−r** denies them permission to read or copy the file. Check this with the **ls −l** command:

```
$ chmod go-r display<CR>
$ ls -l<CR>
total 35
-rwx--x--x   1 starship   project   9346  Nov 1   08:06  display
rw-r--r--    1 starship   project   6428  Dec 2   10:24  list
drwx--x--x   2 starship   project     32  Nov 8   15:32  tools
$
```

Permissions on Directories

You can use the **chmod** command to grant or deny permission for directories as well as files. Specify a directory name instead of a file name on the command line.

However, consider the impact on various system users of changing permissions for directories. For example, you grant read permission for a directory to yourself (**u**), members of your group (**g**), and other system users (**o**). Every user who has access to the system will be able to read the names of the files contained in that directory by running the **ls −l** command. Similarly, granting write permission allows the designated users to create new files in the directory and remove existing ones. Granting permission to execute the directory allows designated users to move to that directory (and make it their current directory) by using the **cd** command.

Using Octal Numbers

There are two methods by which the **chmod** command can be executed. The method described above, in which symbols such as **r**, **w**, and **x** are used to specify permissions, is called the *symbolic* method.

An alternative method is the *octal* method. Its format requires you to specify permissions using three octal numbers, ranging from 0 to 7. (The octal number system is different from the decimal system that we typically use on a day-to-day basis.) To learn how to use the octal method, see the **chmod**(1) page in the *User's Reference Manual*.

Table 3-15 summarizes the syntax and capabilities of the **chmod** command.

Table 3-15. Summary of the **chmod** Command

COMMAND RECAP		
chmod – change permission modes for files (and directories)		
Command	**Instruction**	**Arguments**
chmod	**who + − permission**	*filename(s)* *directoryname(s)*
Description:	**chmod** gives (+) or removes (−) permission to read, write, and execute files for three categories of system users: **user** (you), **group** (members of your group), and **other** (all other users able to access the system on which you are working).	
Remarks:	The instruction set can be represented in either octal or symbolic terms.	

Advanced Commands

Use of the commands already introduced will increase your familiarity with the file system. As this familiarity increases, so might your need for more sophisticated information processing techniques when working with files. This section introduces three commands that provide these techniques:

diff finds differences between two files

grep searches for a pattern in a file

sort sorts and merges files

For additional information about these commands refer to the *User's Reference Manual.*

Comparing Files: diff Command

The **diff** command locates and reports all differences between two files and tells you how to change the first file so that it is a duplicate of the second. The basic format for the command is:

diff *file1 file2*<**CR**>

If *file1* and *file2* are identical, the system returns a prompt to you. If they are not, the **diff** command instructs you on how to change the first file so it matches the second by using **ed** (line editor) commands. (See Chapter 5 for details about the line editor.) The operating system flags lines in *file1* (to be changed) with the < (less than) symbol, and lines in *file2* (the model text) with the > (greater than) symbol.

For example, you execute the **diff** command to identify the differences between the files **johnson** and **mcdonough**. The **mcdonough** file contains the same letter that is in the **johnson** file, with appropriate changes for a different recipient. The **diff** command identifies those changes as follows:

```
3,6c3,6
< Mr. Ron Johnson
< Layton Printing
< 52 Hudson Street
< New York, N.Y.
---
> Mr. J.J. McDonough
> Ubu Press
> 37 Chico Place
> Springfield, N.J.
9c9
< Dear Mr. Johnson:
---
> Dear Mr. McDonough:
```

The first line of output from **diff** is :

```
3,6c3,6
```

This means that if you want **johnson** to match **mcdonough**, you must change (c) lines 3 through 6 in **johnson** to lines 3 through 6 in **mcdonough**. The **diff** command then displays both sets of lines.

If you make these changes (using a text editor such as **ed** or **vi**), the **johnson** file will be identical to the **sanders** file. Remember, the **diff** command identifies differences between specified files.

Table 3-16 summarizes the syntax and capabilities of the **diff** command.

Table 3-16. Summary of the **diff** Command

Command Recap **diff** – finds differences between two files		
Command	**Options**	**Arguments**
diff	available*	*file1 file2*
Description:	The **diff** command reports which lines are different in two files and what you must do to make the first file identical to the second.	
Remarks:	Instructions on how to change a file to bring it into agreement with another file are line editor (**ed**) commands: **a** (append), **c** (change), and **d** (delete). Numbers given with **a**, **c**, or **d** show the lines to be modified. Also used are the symbols < (showing a line from the first file) and > (showing a line from the second file).	

* See the **diff**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Searching a File for a Pattern: grep Command

You can instruct the operating system to search through a file for a specific word, phrase, or group of characters by executing the command **grep** (short for **g**lobally search for a **r**egular **e**xpression and **p**rint). Put simply, a regular expression is any pattern of characters (a word, a phrase, or an equation) that you specify.

The basic format for the command line is:

grep *pattern filelist*<**CR**>

For example, to locate any lines that contain the word **automation** in the file **johnson**, type:

grep automation johnson<**CR**>

The system responds:

```
and office automation software.
```

The output consists of all the lines in the file **johnson** that contain the pattern for which you were searching (automation).

If the pattern contains multiple words or any character that conveys special meaning to the operating system (e.g., **$**, **|**, *****, **?**) the entire pattern must be enclosed in apostrophes. (For an explanation of the special meaning for these and other characters see *Metacharacters* in Chapter 7.) For example, you want to locate the lines containing the pattern **office automation**. Your command line and the system's response will read:

```
$ grep 'office automation' johnson<CR>
and office automation software.
$
```

But what if you cannot recall which letter contained a reference to office automation; your letter to Mr. Johnson or the one to Mrs. Sanders? Type the following command line to find out:

```
$ grep 'office automation' johnson sanders<CR>
johnson:and office automation software.
$
```

The output tells you that the pattern **office automation** is found once in the **johnson** file.

In addition to the **grep** command, the operating system provides variations of it called **egrep** and **fgrep**, along with several options that enhance the searching powers of the command. See the **grep**(1), **egrep**(1), and **fgrep**(1) pages in the *User's Reference Manual* for further information about these commands.

Table 3-17 summarizes the syntax and capabilities of the **grep** command.

Table 3-17. Summary of the **grep** Command

Command Recap **grep** – searches a file for a pattern		
Command	**Options**	**Arguments**
grep	available*	*pattern filelist*
Description:	The **grep** command searches through specified files for lines containing a pattern, then prints the lines on which it finds the pattern. When you specify more than one file, the name of the file in which the pattern is found is also reported.	
Remarks:	If the pattern you give contains multiple words or special characters, enclose the pattern in apostrophes on the command line.	

* See the **grep**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Sorting and Merging Files: sort Command

The operating system provides an efficient tool called **sort** for sorting and merging files. The format for the command line is:

sort *filelist*<**CR**>

The **sort** command causes lines in the specified files to be sorted and merged in the following order:

1. Lines beginning with numbers are sorted by digit and listed before lines beginning with letters.
2. Lines beginning with uppercase letters are listed before lines beginning with lowercase letters.
3. Lines beginning with symbols (e.g., *, **%**, or @) are sorted on the basis of the symbol's ASCII value.

For example, you have two files, **group1** and **group2**, each containing a list of names. You want to sort each list alphabetically and then interleave the two lists into one. First, display the contents of the files by executing the **cat** command on each.

```
$ catgroup1<CR>
Smith, Allyn
Jones, Barbara
Cook, Karen
Moore, Peter
Wolf, Robert
$ catgroup2<CR>
Frank, M. Jay
Nelson, James
West, Donna
Hill, Charles
Morgan, Kristine
$
```

(Instead of printing these two files individually, you could have requested both files on the same command line. If you had typed **cat group1 group2** and pressed the RETURN key, the output would have been the same.)

Now sort and merge the contents of the two files by executing the **sort** command. The output of the **sort** program prints on the screen unless you specify otherwise:

```
$ sort group1 group2<CR>
Cook, Karen
Frank, M. Jay
Hill, Charles
Jones, Barbara
Moore, Peter
Morgan, Kristine
Nelson, James
Smith, Allyn
West, Donna
Wolf, Robert
$
```

In addition to combining simple lists as in the example, the **sort** command can rearrange lines and parts of lines (called fields) according to a number of other specifications you designate on the command line. The possible specifications are complex and beyond the scope of this text. Refer to the *User's Reference Manual* for a full description of available options.

Table 3-18 summarizes the syntax and capabilities of the **sort** command.

Table 3-18. Summary of the **sort** Command

Command Recap **sort** – sorts and merges files		
Command	**Options**	**Arguments**
sort	available*	*filelist*
Description:	The **sort** command sorts and merges lines from a file or files you specify and displays its output on your screen.	
Remarks:	If no options are specified on the command line, lines are sorted and merged in the order defined by the ASCII representations of the characters in the lines.	

* See the **sort**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Printing Files

This section introduces methods of preparing files to be printed (with the **pr** command) and of printing them (with the **lp** command).

Printing a File: pr Command

The **pr** command is used to prepare files for printing. It supplies titles and headings, paginates, and prints a file, in any of various page lengths and widths, on your screen.

You have the option of requesting that the command print its output on another device, e.g., a line printer (read the discussion of the **lp** command in this section). You can also direct the output of **pr** to a different file (see the section *Input and Output Redirection* in Chapter 7).

If you choose not to specify any of the available options, the **pr** command produces output in a single column that contains 66 lines per page and is preceded by a short heading. The heading consists of five lines: two blank lines; a line containing the date, time, file name, and page number; and two more blank lines. The body of the text is followed by five blank lines.

The **pr** command is often used with the **lp** command to provide a paper copy of text as it was entered into a file. (See the section on the **lp** command for details.) However, you can also use the **pr** command to print the contents of a file on your screen. For example, to review the contents of the file **johnson** in the sample file system, type:

```
$ pr johnson<CR>
```

The following screen gives an example of output from this command:

```
$ prjohnson<CR>


Mar  5 15:43 1986 johnson Page 1


March 5, 1986

Mr. Ron Johnson
Layton Printing
52 Hudson Street
New York, N.Y.


Dear Mr. Johnson:

I enjoyed speaking with you this morning
about your company's plans to automate
your business.
Enclosed please find
the material you requested
about AB&C's line of computers
and office automation software.

If I can be of further assistance to you,
please don't hesitate to call.

Yours truly,

John Howe
.
.
$
```

The periods after the last line in the file represent the remaining lines (all blank in this case) that **pr** added to the output so that each page would contain a total of 66 lines. If you are working on a video display terminal, which allows you to view 24 lines at a time, the entire 66 lines of the displayed file prints rapidly without pause. This means that the first 42 lines will roll off the top of your screen, making it impossible for you to read them unless you have the ability to roll back a screen or two. However, if the file you are examining is particularly long, even this ability may not be sufficient to allow you to read the file.

In such cases, type <^> **(CTRL-s)** to interrupt the flow of printing on your screen. When you are ready to continue, type <^**q**> (**CTRL-q**) to resume printing.

Table 3-19 summarizes the syntax and capabilities of the **pr** command.

Table 3-19. Summary of the **pr** Command

Command Recap **pr** – print contents of a file		
Command	**Options**	**Arguments**
pr	available*	*filelist*
Description:	The **pr** command produces a copy of a file(s) on your screen unless you specify otherwise. It prints the text of the file(s) on 66 line pages, and places five blank lines at the bottom of each page and a five-line heading at the top of each page. The heading includes: two blank lines; a line containing the date, time, file name, and page number; and two additional blank lines.	
Remarks:	If a specified file exists, its contents display; if not, the message `pr: can't open` *filename* prints. The **pr** command is often used with the **lp** command to produce a paper copy of a file. It can also be used to review a file on a video display terminal. To stop and restart the printing of a file on a terminal, type <^**s**> and <^**q**>, respectively.	

* See the **pr**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

The LP Print Service

You can perform various printing tasks by using a set of the operating system software tools called the LP print service. You can make requests for print jobs, change or cancel those requests, enable or disable printers, and obtain information about the printers available to you by using five commands associated with the LP print service: **lp, cancel, lpstat, enable,** and **disable**. This section explains how to use these commands to accomplish these tasks.

The function of each print service command is shown in Table 3-20.

Table 3-20. Print Commands and Their Functions

Command	Function
lp	requests a paper copy of a printable file from a printer.
cancel	cancels a request for a paper copy of a file.
lpstat	displays information on the screen about the current status of the LP print service.
enable	activates the printer(s) specified so jobs that are requested through the **lp** command can be printed.
disable	deactivates the printer(s) specified so jobs that are requested through the **lp** command can no longer be printed.

NOTE

To prevent printer settings from being changed, only ordinary files that are printable should be used by **lp** (i.e., not executable or data files).

Requesting a Paper Copy of a File: lp Command

Some terminals have built-in printers that allow you to get paper copies of files. If you have such a terminal, you can get a paper copy of your file simply by turning on the printer and executing the **cat** or **pr** command.

If you are using a video display terminal however, you will need a printer to obtain a paper copy of a file. The **lp** command (originally named for "line printer") allows you to request a print job from a printer. To request a simple print job, enter the command:

lp *filename*<**CR**>

where *filename* is the name of the file you want to have printed. For example, to request that the file **johnson** be printed, type:

lp johnson<**CR**>

The system responds with the name (or type) of the printer on which the file is being printed and an identification (ID) number for your request:

```
$ lp johnson<CR>
request id is laser-6885  (1 file)
$
```

This system response shows that your job will be printed on a printer named "laser" (the default printer for this system), has a request ID number of laser-6885, and includes of one file.

The options available with the **lp** command allow you to specify the following for your print job: a specific printer or class of printers (referred to here as "destination"), special print modes (e.g., landscape or portrait), page size and pitch settings, a list of pages to be printed and the number of copies to be made, queue priority, forms (instead of blank paper), character sets and print wheels, content type, continuous printing of files (without breaks between separate files), banner-page options, and messages from the **lp** command.

Select a Print Destination

The term "print destination" refers to any device that your system administrator has defined to be a printer (e.g., **bif2**) or class of printers (e.g., **bif**). The **–d** *dest* (short for destination) option on the command line causes your file to be printed at the destination specified in the *dest* argument, as long as a printer is available and capable of meeting your specifications for the job. In this example, a request is made to have a file called **memo** printed on printer3:

```
$ lp –d printer3 memo<CR>
```

Canceling a Request: cancel Command

To cancel a request to a printer, type the command **cancel** and specify the request ID number. For example, to cancel the printing of the file **letters** (request ID laser-6885), type:

```
$ cancel laser-6885<CR>
```

Note that you can cancel only your own requests.

Getting Printer Status: lpstat Command

To check the status of a printer job that is in progress or to get its request ID number, execute the **lpstat** command. The **-d** option lists current output requests; the **-t** provides a complete listing. This command also provides a complete listing of every printer available on your system. Which printers are available to you depends on your operating system facility. Ask your System Administrator for the names of available line printers, or type the following command:

```
$ lpstat -v<CR>
```

Enabling and Disabling a Printer: enable and disable

NOTE

Whether or not you, as a computer user, are able to issue the commands to enable and disable printers depends on your System Administrator. Because these functions are administrative, it is left to the discretion of the System Administrator to decide whether or not to make the **enable** and **disable** commands available to users.

Before a printer is able to start printing files requested through the **lp** command, it must be activated. You can activate a printer by issuing the **enable** command with one argument: one printer or a list of printers:

$ enable *printer1 printer2 printer3*<**CR**>

You can verify that you have enabled a printer by requesting a status report for it.

If you want a printer to stop printing jobs, you must deactivate it by issuing the **disable** command:

$ disable printer1<**CR**>

Tables 3-21 and 3-22 summarize the syntax and capabilities of the **lp** and **lpstat** commands, respectively.

Table 3-21. Summary of the **lp** Command

Command Recap		
lp – request paper copy of a file from a printer		
Command	**Options**	**Arguments**
lp	*(as listed)*	*file(s)*

Description: The **lp** command requests that specified files be printed by a printer, thus providing paper copies of the contents.

Options:

–d *dest*	Allows you to choose *dest* as the printer or class of printers to produce the paper copy. You do not have to use this option if the administrator has set a default destination or if you have set the LPDEST environment variable.
–y *mode*	Requests special printing modes, such as portrait or landscape. (This option requires a special filter; check with your system administrator to find out whether your system has one.)
–o *option*	Defines page dimensions: length and width, number of lines per inch, and number of characters per inch. (**–o** performs other tasks, too; see **lp**(1) in the *User's Reference Manual*.)
–P *pages*	Specifies subset of pages to be printed. (This option requires a special filter; check with your system administrator to find out whether your system has an appropriate filter.)
–n *copies*	Specifies number of copies to be made.

Table 3-21. Summary of the **lp** Command (cont'd)

Command Recap		
lp – request paper copy of a file from a printer		
Command	**Options**	**Arguments**
lp	*(as listed)*	*file(s)*
	–f *form*	Specifies pre-printed form on which files are to be printed.
	–S *char_set*	Specifies character set or print wheel to be used.
	–T *type*	Specifies content type of print request.
	–w	Notifies you by screen message when print job is complete.
	–m	Notifies you by mail when print job is complete.
	–i *req_id*	Allows you to change a print request already issued (but not yet printed).
	–q *level*	Allows you to specify a priority level for your job request.
Remarks:	Check with your System Administrator for information on additional and/or different commands for printers that may be available at your location.	

Table 3-22. Summary of the **lpstat** Command

Command Recap		
lpstat – display information about status of LP print service		
Command	**Options**	**Arguments**
lp	*(as listed)*	*file(s)*

Description: The **lpstat** command reports the status ov print requests, printers, and the LP request scheduler, and provides other information related to the status of the print service.

Options:

–a [*list*]	Reports whether print requests are being accepted by specified printers or classes of printers.
–c [*list*]	Lists the names of printer classes and members of each.
–d	Shows the default destination for your LP print service.
–f [*list*] [**–l**]	Verifies that the forms named in *form-list* are recognized by the LP print service. The **–l** option lists the form descriptions.
–o [*list*] [**–l**]	Reports the status of print requests. *List* may include names of printers or printer classes, or request ids.
–p [*list*] [**–D**] [**–l**]	Reports the status of printers named in *list*. The **–D** option adds a description of each printer, and **–l** requests a full description of each printer's configuration.

Table 3-22. Summary of the **lpstat** Command (cont'd)

Command Recap **lpstat** – display information about status of LP print service		
Command	**Options**	**Arguments**
lp	(*as listed*)	*file(s)*
	–r	Reports the status of the LP request scheduler.
	–s	Prints a status summary of the whole LP print service.
	–S [*list*] [**–l**]	Verifies that the character sets or print wheels specified in *list* are recognized by the LP print service. The **–l** option requests a list of printers that can handle each character set and print wheel.
	–t	Prints all status information.
	–u [*list*]	Reports status of users' print requests. *List* is a list of login names.
	–v [*list*]	Lists printers and the pathnames of the devices associated with them. *List* is a list of printer names.
Remarks:	In each case where *list* is indicated, you can also specify **all**.	

4 Overview of the Tutorials

Introduction

This chapter provides an overview of the tutorials in the next four chapters. Specifically, text editing, working in the shell, and electronic mail. Text editing is covered in Chapter 5, *Line Editor Tutorial (ed)*, and Chapter 6, *Screen Editor Tutorial (vi)*. Working and programming in the shell is taught in Chapter 7, *Shell Tutorial*, and **mail** and **mailx** are covered in Chapter 8, *Electronic Mail Tutorial*.

Text Editing

Using the file system is a way of life in a SYSTEM V/88 environment. This section will teach you how to create and modify files with a software tool called a text editor. The section begins by explaining what a text editor is and how it works. Then it introduces two types of text editors supported on the operating system: the line editor, **ed**, and the screen editor, **vi** (short for visual editor). A comparison of the two editors is also included. For detailed information about **ed** and **vi**, refer to Chapters 5 and 6.

What Is a Text Editor?

Whenever you revise a letter, memo, or report, you must perform one or more of the following tasks: insert new or additional material, delete unneeded material, transpose material (sometimes called cutting and pasting), and, finally, prepare a clean, corrected copy. Text editors perform these tasks at your direction, making writing and revising text much easier and quicker than if done by hand.

The operating system text editors, like the operating system shell, are interactive programs; they accept your commands and then perform the requested functions. From the shell's point of view, the editors are executable programs.

A major difference between a text editor and the shell, however, is the set of commands that each recognizes. A text editor has its own distinct set of commands that allow you to create, move, add, and delete text in files, as well as acquire text from other files.

How Does a Text Editor Work?

To understand how a text editor works, you need to understand the environment created when you use an editing program and the modes of operation understood by a text editor.

Text Editing Buffers

When you use a text editor to create a new file or modify an existing one, you first ask the shell to put the editor in control of your computing session. As soon as the editor takes over, it allocates a temporary work space called the *editing buffer*; any information you enter while editing a file is stored in this buffer where you can modify it.

Because the buffer is a temporary work space, any text you enter and any changes you make to it are only stored in temporary memory. The buffer and its contents will exist only as long as you are editing. If you want to save the file, you must tell the text editor to write the contents of the buffer into a file. The file is then stored in the computer's long term disk memory. If you do not, the buffer's contents will disappear when you leave the editing program. To prevent this from happening, the text editors send you a reminder to write your file if you attempt to end an editing session without doing so.

NOTE

> If you have made a critical mistake or are unhappy with the edited version, you can choose to leave the editor without writing the file. By doing so, you leave the original file intact; the edited copy disappears.

Regardless of whether you are creating a new file or updating an existing one, the text in the buffer is organized into lines. A line of text is simply a series of characters that appears horizontally across the screen and is ended when you press the RETURN key. Occasionally, files may contain a line of text that is too long to fit on the terminal screen. Some terminals automatically display the continuation of the line on the next row of the screen; others do not.

Modes of Operation

Most text editors are capable of understanding two modes of operation: command mode and text input mode. When you begin an editing session, you will be placed automatically in command mode. In this mode you can move around in a file, search for patterns in it, or change existing text. However, you cannot create text while you are in command mode. To do this you must be in text input mode. While you are in this mode, any characters you type are placed in the buffer as part of your text file. When you have finished entering text and want to run editing commands again, you must return to command mode.

Because a typical editing session involves moving back and forth between these two modes, you may sometimes forget which mode you are working in. You may try to enter text while in command mode or to enter a command while in input mode. This is something even experienced users do from time to time. It will not take long to recognize your mistake and determine the solution after you complete the tutorials in Chapters 5 and 6.

Line Editor

The line editor, accessed by the **ed** command, is a fast, versatile program for preparing text files. It is called a line editor because it manipulates text on a line-by-line basis. This means you must specify, by line number, the line containing the text you want to change. Then **ed** prints the line on the screen where you can modify it.

This text editor provides commands with which you can change lines, print lines, read and write files, and enter text. In addition, you can invoke the line editor from a shell program; something you cannot do with the screen editor. (See Chapter 7 for information on basic shell programming techniques.)

The line editor (**ed**) works well on video display terminals and paper printing terminals. It will also accommodate a slow-speed telephone line. (The visual editor, **vi**, can be used only on video display terminals.) See Chapter 5, *Line Editor Tutorial (ed)*, for instructions on this editing tool. Also see Appendix C for a summary of line editor commands.

Screen Editor

The screen editor, accessed by the **vi** command, is a display-oriented, interactive software tool. It allows you to view, a page at a time, the file you are editing. This editor works most efficiently when used on a video display terminal operating at 1200 or higher baud.

You modify a file (by adding, deleting, or changing text) by positioning the cursor at the point on the screen where the modification is to be made, then make the change. The screen editor immediately displays the results of your editing; you can see the change you made in the context of the surrounding text. Because of this feature, the screen editor is considered more sophisticated than the line editor.

In addition, the screen editor offers a choice of commands. For example, a number of screen editor commands allow you to move the cursor around a file. Other commands scroll the file up or down on the screen. Still other commands allow you to change existing text or to create new text. In addition to its own set of commands, the screen editor can access line editor commands.

The trade-off for the screen editor's speed, visual appeal, efficiency, and power is the heavy demand it places on the computer's processing time. Every time you make a change, no matter how simple, **vi** must update the screen. See Chapter 6, *Screen Editor Tutorial (vi)*, for instructions on how to use this editor. Appendix D contains a summary of screen editor commands, and Table 4-1 compares the features of the line editor (**ed**) and the screen editor (**vi**).

Table 4-1. Comparison of Line and Screen Editors (**ed** and **vi**)

Feature	Line Editor (ed)	Screen Editor (vi)
Recommended terminal type	Video display or paper-printing.	Video display.
Speed	Accommodates high- and low-speed data transmission lines.	Works best via high-speed data transmission lines (1200+ baud).
Versatility	Can be specified to run from shell scripts as well as used during editing sessions.	Must be used interactively during editing sessions.
Sophistication	Changes text quickly. Uses comparatively small amounts of processing time.	Changes text easily. However, can make heavy demands on computer resources.
Power	Provides a full set of editing commands. Standard operating system text editor.	Provides its own editing commands and recognizes line editor commands as well.
Advantages	There are fewer commands you must learn to use **ed**.	Allows you to see the effects of your editing in the context of a page of text, immediately. (When you use the **ed** editor, making changes and viewing the results are separate steps.)

The Shell

Every time you log in to the operating system you start communicating with the shell, and continue to do so until you log off the system. However, while you are using a text editor, your interaction with the shell is suspended; it resumes as soon as you stop using the editor.

The shell is much like other programs, except that instead of performing one job, as **cat** or **ls** does, it is central to your interactions with the operating system. The shell's primary function is to act as a command interpreter between you and the computer system. As an interpreter, the shell translates your requests into language the computer understands, calls requested programs into memory, and executes them.

This section introduces methods of using the shell that enhance your ability to use system features. In addition to using it to run a single program, you may also use the shell to:

- interpret the name of a file or a directory you enter in an abbreviated way using a type of shell shorthand
- redirect the flow of input and output of the programs you run
- execute multiple programs simultaneously or in a pipeline format
- tailor your computing environment to meet your individual needs

In addition to being the command language interpreter, the shell is a programming language. For detailed information on how to use the shell as a command interpreter and a programming language, see Chapter 7.

Customizing Your Computing Environment

This section deals with another control provided by the shell: your environment. When you log in to the operating system, the shell automatically sets up a computing environment for you. The environment set up by the shell includes these variables:

HOME	your login directory
LOGNAME	your login name
PATH	route the shell takes to search for executable files or commands (typically **PATH=:/bin:/usr/bin**)

The **PATH** variable tells the shell where to look for the executable program invoked by a command. Therefore, it is used every time you issue a command. If you have executable programs in more than one directory, you will want all of them to be searched by the shell to make sure every command can be found.

You can use the default environment supplied by your system or you can tailor an environment to meet your needs. If you choose to modify any part of your environment, you can use either of two methods to do so. If you want to change a part of your environment only for the duration of your current computing session, specify your changes in a command line (see Chapter 7 for details). However, if you want to use a different environment (not the default environment) regularly, you can specify your changes in a file that will set up the desired environment for you automatically every time you log in. This file must be called **.profile** and must be located in your home directory.

The **.profile** typically performs some or all the following tasks: checks for mail; sets data parameters, terminal settings, and tab stops; assigns a character or character string as your login prompt; and assigns the erase and kill functions to keys. You can define as few or as many tasks as you want in your **.profile**. You can also change parts of it at any time.

Now check to see whether or not you have a **.profile**. If you are not already in your home directory, **cd** to it. Then examine your **.profile** by issuing this command:

```
cat .profile
```

If you have a **.profile**, its contents appears on your screen. If you do not have a **.profile**, you can create one with a text editor. For instructions on creating and modifying a **.profile**, see *Modifying Your Login Environment* in Chapter 7.

Programming in the Shell

The shell is not only the command language interpreter; it is also a command level programming language. This means that instead of always using the shell strictly as a liaison between you and the computer, you can also program it to repeat sequences of instructions automatically. To do this, you must create executable files containing lists of commands. These files are called *shell procedures* or *shell scripts*. Once you have a shell script for a particular task, you can request that the shell read and execute the contents of the script whenever you want to perform that task.

Like other programming languages, the shell supports such features as variables, control structures, subroutines, and parameter passing. These features enable you to create your own tools by linking together system commands.

For example, you can combine three operating system programs (**date**, **who**, and **wc** commands) into a simple shell script called **users** that tells you the current date and time, and how many users are working on your system. If you use the **vi** editor (described in Chapter 6) to create your script, you can follow this procedure. First, create the file **users** with the editor by typing:

vi users<CR>

The editor will draw a blank page on your screen and wait for you to enter text.

```
cursor
~
~
~
~
~
~
~
~
~
"users" [New file]
```

Enter the three operating system commands on one line:

date; who | wc –l

Then write and quit the file. Make **users** executable by adding execute permission with the **chmod** command:

chmod ug+x users<CR>

Now try running your new command. The following screen shows the kind of output that displays:

```
$ users<CR>
Wed Mar 1 16:40:12 EST 1989
       4
$
```

The output tells you that four users were logged in on the system when you typed the command at 16:40 on Wednesday, March 1, 1989.

For step-by-step instructions on writing shell scripts and information about more sophisticated shell programming techniques, see Chapter 7, *Shell Tutorial*.

Communicating Electronically

As an operating system user, you can send messages or transmit information stored in files to other users who work on your system or another operating system. To do so, you must be logged in on an operating system that is capable of communicating with the operating system to which you want to send information. The command you use to send information depends on what you are sending. This guide introduces you to these communication programs:

mail
allows you to send messages or files to other operating system users, using their login names as addresses. It also allows you to receive messages sent by other users. **mail** holds messages and lets the recipient read them at their convenience.

mailx
is a sophisticated, more powerful version of **mail**. It offers a number of options for managing the electronic mail you send and receive.

uucp
is used to send files from one operating system to another. You can use **uucp** to send a file to a directory you specify on a remote computer. When the file has been transferred, the owner of the directory is notified of its arrival by **mail**.

uuto/uupick
are used to send and retrieve files. You can use the **uuto** command to send files to a public directory. Once they are available, the recipient is notified by mail that the files have arrived. The recipient then can use the **uupick** command to copy the files from the public directory to a directory of choice.

uux
lets you execute commands on a remote computer. It gathers files from various computers, executes the specified command on these files, and sends the standard output to a file on the specified computer.

Chapter 8 offers tutorials on each of these.

Programming in the System

The operating system provides a powerful and convenient environment for programming and software development using the C programming language and other Independent Software Vendor (ISV) software. The operating system also provides some sophisticated tools designed to make software development easier and more systematic.

For information on the available operating system programming languages, see the *Programmer's Guide*. Besides supplementing texts on programming languages, the guide provides tutorials on the following four tools:

SCCS	Source Code Control System
make	maintains programs
lex	generates programs for simple lexical tasks
yacc	generates parser programs

5 Line Editor Tutorial (ed)

Introducing the Line Editor

This chapter is a tutorial on the line editor, **ed**. **ed** is versatile and requires little computer time to perform editing tasks. It can be used on any type of terminal. The examples of command lines and system responses in this chapter will apply to your terminal, whether it is a video display terminal or a paper printing terminal. The **ed** commands can be typed in at your terminal or they can be used in a shell program. (See Chapter 7, *Shell Tutorial*.)

ed is a line editor; during editing sessions it is always pointing at a single line in the file called the current line. When you access an existing file, **ed** makes the last line the current line so you can start appending text easily. Unless you specify the number of a different line or range of lines, **ed** does each command you issue on the current line. In addition to letting you change, delete, or add text on one or more lines, **ed** allows you to add text from another file to the buffer.

During an editing session with **ed**, you are altering the contents of a file in a temporary buffer, where you work until you have finished creating or correcting your text. When you edit an existing file, a copy of that file is placed in the buffer and your changes are made to this copy. The changes have no effect on the original file until you instruct **ed**, by using the write **w** subcommand, to move the contents of the buffer into the file.

After you read through this tutorial and tried the examples and exercises, you will have a good working knowledge of **ed**. The following topics are covered:

- entering the line editor **ed**, creating text, writing the text to file, and quitting **ed**
- addressing particular lines of the file and displaying lines of text
- deleting text
- substituting new text for old text
- using special characters as shortcuts in search and substitute patterns
- moving text around in the file, as well as other useful commands and concepts.

The commands are discussed in individual sections and reviewed at the end of each section. In Appendix C, you will find a summary of all **ed** commands introduced here.

At the end of some sections, exercises are given so you can experiment with the commands. The answers to all exercises are at the end of this chapter.

Getting Started

The best way to learn **ed** is to log in to the operating system and try the examples as you read this tutorial. Do the exercises; do not be afraid to experiment. As you experiment and try out **ed** commands, you will learn a fast and versatile method of text editing.

In this section you will learn the commands used to:

- enter **ed**
- append text
- move up or down in the file to display a line of text
- delete a line of text
- write the buffer to a file
- quit **ed**

Entering ed

To enter the line editor, type **ed** and a file name:

ed *filename*<**CR**>

Choose a name that reflects the contents of the file. If you are creating a new file, the system responds with a question mark and the file name:

```
$ ed new-file<CR>
?new-file
```

When you edit an existing file, **ed** responds with the number of characters in the file:

```
$ ed old-file<CR>
235
```

Creating Text

The editor receives two types of input, editing commands and text, from your terminal. To avoid confusing them, **ed** recognizes two modes of editing work: command mode and text input mode. When you work in command mode, any characters you type are interpreted as commands. In input mode, any characters you type are interpreted as text to be added to a file.

Whenever you enter **ed**, you are put into command mode. To create text in your file, change to input mode by typing **a** (for append), on a line by itself, and pressing the **RETURN** key:

a<CR>

Now you are in input mode; any characters you type from this point are added to your file as text. Be sure to type **a** on a line by itself; if you do not, the editor will not execute your command.

After you have finished entering text, type a "." (period) on a line by itself. This takes you out of the text input mode and returns you to the command mode. Now you can enter **ed** other commands.

The following example shows how to enter **ed**, create text in a file called **try-me**, and quit text input mode with a period.

```
$ ed try-me<CR>
? try-me
a<CR>
This is the first line of text.<CR>
This is the second line,<CR>
and this is the third line.<CR>
.<CR>
```

Notice that **ed** does not give a response to the period; it just waits for a new command. If **ed** does not respond to a command, you may have forgotten to type a period after entering text and may still be in text input mode. Type a period and press the RETURN key at the beginning of a line to return to command mode. Now you can execute editing commands. For example, if you have added some unwanted characters or lines to your text, you can delete them once you have returned to command mode.

Displaying Text

To display a line of a file, type **p** (for print) on a line by itself. The **p** command prints the current line, i.e., the last line on which you worked. Continue with the previous example. You have just typed a period to exit input mode. Now type the **p** command to see the current line.


```
$ ed try-me<CR>
? try-me
a<CR>
This is the first line of text.<CR>
This is the second line,<CR>
and this is the third line.<CR>
.<CR>
p<CR>
and this is the third line.
```

You can print any line of text by specifying its line number (also known as the address of the line); the address of the first line is 1; the second, 2. For example, to print the second line in the file **try-me**, type:

```
2p<CR>
This is the second line,
```

You can also use line addresses to print a span of lines by specifying the addresses of the first and last lines of the section you want to see, separated by a comma. For example, to print the first three lines of a file, type:

1,3p<CR>

You can even print the entire file this way. For example, you can display a 20 line file by typing **1,20p**. If you do not know the address of the last line in your file, you can substitute a **$** sign, **ed** symbol for the address of the last line. (These conventions are discussed in the section *Line Addressing*.)

```
1,$p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
```

If you forget to quit text input mode with a period, you will add text that you do not want. Try to make this mistake. Add another line of text to your **try-me** file; enter the **p** command without quitting text input mode; then quit text input mode and print the entire file.

```
p<CR>
and this is the third line.
a<CR>
This is the fourth line.<CR>
p<CR>
.<CR>
1,$p<CR>
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line.
p
```

The next section explains how to delete the unwanted line.

Deleting a Line of Text

To delete text, you must be in the command mode of **ed**. Typing **d** deletes the current line. Try this command on the last example to remove the unwanted line containing p. Display the current line (**p** command), delete it (**d** command), and display the remaining lines in the file (**p** command). Your screen should look like the one that follows:

```
p<CR>
p
d<CR>
1,$p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
```

ed does not send you any messages to confirm that you have deleted text. The only way you can verify that the **d** command has succeeded is by printing the contents of your file with the **p** command. To receive verification of your deletion, you can put the **d** and **p** together on one command line. If you repeat the previous example with this command, your screen should look like the following:

```
p<CR>
p
dp<CR>
This is the fourth line.
```

Moving Up or Down in the File

To display the line below the current line, press the **RETURN** key while in command mode. If there is no line below the current line, **ed** responds with a **?** and continues to treat the last line of the file as the current line. To display the line above the current line, press the minus key (−). The following screen provides examples of how both of these commands are used:

```
p<CR>
This is the fourth line.
-<CR>
and this is the third line.
-<CR>
This is a second line,
-<CR>
This is the first line of text.
<CR>
This is a second line,
<CR>
and this is the third line.
```

By typing −<**CR**> or <**CR**>, you can display a line of text without typing the **p** command. These commands are also line addresses. Whenever you type a line address and do not follow it with a command, **ed** assumes that you want to see the line you have specified. Experiment with these commands: create some text, delete a line, and display your file.

Saving the Buffer Contents in a File

As discussed earlier, during an editing session, the system holds your text in a temporary storage area called a buffer. When you have finished editing, you can save your work by writing it from the temporary buffer to a permanent file. This stores your file data in the computers long term memory disk. By writing to a file, you are putting a copy of the contents of the buffer into the file. The text in the buffer is not disturbed, and you can make further changes to it.

NOTE

It is a good idea to write the buffer text into your file frequently. If an interrupt occurs (e.g., as an accidental loss of power to your terminal), you may lose the material in the buffer, but you will not lose the copy written to your file.

To write your text to a file, enter the **w** command. You do not need to specify a file name; simply type **w** and press the **RETURN** key. If you have just created new text, **ed** creates a file for it with the name you specified when you entered the editor. If you edited an existing file, the **w** command writes the contents of the buffer to that file by default.

If you prefer, you can specify a new name for your file as an argument on the **w** command line. Be careful not to use the name of a file that already exists unless you want to replace its contents with the contents of the current buffer. **ed** does not warn you about an existing file; it simply overwrites that file with your buffer contents.

For example, if you decide you would prefer the **try-me** file to be called **stuff**, you can rename it:

```
$ ed try-me<CR>
? try-me
a<CR>
This is the first line of text.<CR>
This is the second line,<CR>
and this is the third line.<CR>
.
w stuff <CR>
85
```

Notice the last line of the screen. This is the number of characters in your text. When the editor reports the number of characters in this way, the write command has succeeded.

Quitting the Editor

When you have completed editing your text, write it from the buffer into a file with the **w** command. Then leave the editor and return to the shell by typing **q** (for quit):


```
w<CR>
85
q<CR>
$
```

The system responds with a shell prompt. At this point the editing buffer is empty. If you did not execute the **write** command, your text in the buffer has also vanished. If you did not make any changes to the text during your editing session, no harm is done. However, if you did make changes, you could lose your work. Therefore, if you type **q** after changing the file without writing it, **ed** warns you with a **?**. You then have a chance to write and quit.

```
q<CR>
?
w<CR>
85
q<CR>
$
```

If, instead of writing, you insist on typing **q** a second time, **ed** assumes you do not want to write the buffer's contents to your file and returns you to the shell. Your file is left unchanged and the contents of the buffer are erased.

You now know the basic commands for editing and creating a file using **ed**. Table 5-1 summarizes these commands.

Table 5-1. Summary of **ed** Editor Commands

Command	Function
ed *file*	enters **ed** to edit *file*.
a	appends text after the current line.
.	quits text input mode and returns to **ed** command mode.
p	prints text on your terminal.
d	deletes text.
<**CR**>	displays the next line in the buffer (literally, carriage return).
+	displays the next line in the buffer.
–	displays the previous line in the buffer.
w	writes the contents of the buffer to the file.
q	quits **ed** and returns to the shell.

Exercise 1

Answers for all the exercises in this chapter are at the end of the chapter. However, they are not necessarily the only possible correct answers. Any method that enables you to perform a task specified in an exercise is correct, even if it does not match the answer given.

1-1. Enter **ed** with a file named **junk**. Create a line of text containing **Hello World**, write it to the file and quit **ed**.

Now use **ed** to create a file called **stuff**. Create a line of text containing two words, **Goodbye world**, write this text to the file, and quit **ed**.

1-2. Enter **ed** again with the file named **junk**. What was the editor's response? Was the character count for it the same as the character count reported by the **w** command in Exercise 1-1?

Display the contents of the file. Is that your file **junk**?

How can you return to the shell? Try **q** without writing the file. Why do you think the editor allowed you to quit without writing to the buffer?

1-3. Enter **ed** with the file **junk**. Add a line:

Wendy's horse came through the window.

Since you did not specify a line address, where do you think the line was added to the buffer? Display the contents of the buffer. Try quitting the buffer without writing to the file. Try writing the buffer to a different file called **stuff**. Notice that **ed** does not warn you that a file called **stuff** already exists. You have erased the contents of **stuff** and replaced them with new text.

General Format of ed Commands

ed commands have a simple and regular format:

[*address1*[,*address2*]]*command*[*argument*]<**CR**>

The brackets around *address1*, *address2*, and *argument* show that these are optional. The brackets are not part of the command line.

address1,*address2*
: give the position of lines in the buffer. *Address1* through *address2* gives you a range of lines that are affected by the *command*. If *address2* is omitted, the command affects only the line specified by *address1*.

command
: is usually one character and tells the editor what task to perform.

argument
: are those parts of the text that will be modified, or a file name, or another line address.

This format will become clearer to you when you begin to experiment with the **ed** commands.

Line Addressing

A line address is a character or group of characters that identifies a line of text. Before **ed** can execute commands that add, delete, move, or change text, it must know the line address of the affected text. Type the line address before the command:

[*address1*],[*address2*]*command*<**CR**>

Both *address1* and *address2* are optional. Specify *address1* alone to request action on a single line of text; both *address1* and *address2* to request a span of lines. If you do not specify any *address*, **ed** assumes that the line address is the current line.

The most common ways to specify a line address in **ed** are:

- entering line numbers (assuming that the lines of the files are consecutively numbered from 1 to *n*, beginning with the first line of the file)
- entering special symbols for the current line, last line, or a span of lines
- adding or subtracting lines from the current line
- searching for a character string or word on the desired line

You can access one line or a span of lines or make a global search for all lines containing a specified character string. (A character string is a set of successive characters, e.g., a word.)

Line Numbers

ed gives a numerical address to each line in the buffer, e.g., the first line of the buffer is 1, the second line is 2, for each line in the buffer. Any line can be accessed by **ed** with its line address number.

To see how line numbers address a line, enter **ed** with the file **try-me** and type a number:

```
$ ed try-me<CR>
110
1<CR>
This is the first line of text.
3<CR>
and this is the third line.
```

Remember that **p** is the default command for a line address specified without a command. Because you gave a line address, **ed** assumes you want that line displayed on your terminal.

Line numbers frequently change in the course of an editing session. Later in this chapter you will create lines, delete lines, or move a line to a different position. This changes the line numbers of some lines. The number of a specific line is always the current position of that line in the editing buffer. For example, if you add five lines of text between lines 5 and 6, line 6 becomes line 11. If you delete line 5, line 6 becomes line 5.

Symbolic Line Addressing

Addressing the Current Line

The current line is the line most recently acted on by any **ed** command. If you have just entered **ed** with an existing file, the current line is the last line of the buffer. The symbol for the address of the current line is a period. Therefore, you can display the current line by typing a period and pressing the RETURN key.

Try this command in the file **try-me**:

```
$ ed try-me<CR>
110
.<CR>
This is the fourth line.
```

The period is the address. Because a command is not specified after the period, **ed** executes the default command **p** and displays the line found at this address.

To get the line number of the current line, type the following command:

.=<CR>

ed responds with the line number. For example, in the **try-me** file, the current line is 4.

```
.<CR>
This is the fourth line.
.=<CR>
4
```

Addressing the Last Line

The symbolic address for the last line of a file is the **$** sign. To verify that the **$** sign accesses the last line, access the **try-me** file with **ed** and specify this address on a line by itself. (Keep in mind that when you first access a file, your current line is always the last line of the file.)

```
$ ed try-me<CR>
110
.<CR>
This is the fourth line.
$<CR>
This is the fourth line.
```

Remember that the **$** address within **ed** is not the same as the **$** prompt from the shell.

Addressing the Set of All Lines

When used as an address, a comma refers to all lines of a file, from the first through the last line. It is an abbreviated form of the string mentioned earlier that represents all lines in a file, "**1,$**". Try this shortcut to print the contents of **try-me**.

```
,p<CR>
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line.
```

Addressing the Current Line Through the Last Line

The semi-colon represents a set of lines beginning with the current line and ending with the last line of a file. It is equivalent to the ".,$" symbolic address. Try it with the file **try-me**.

```
2p<CR>
This is the second line,
;p<CR>
This is the second line,
and this is the third line.
This is the fourth line.
```

Using Relative Addresses

You may often want to address lines with respect to the current line. You can do this by adding or subtracting a number of lines from the current line with a plus (+) or a minus (−) sign. Addresses derived in this way are called relative addresses.

To experiment with relative line addresses, add several more lines to your file **try-me**, as shown in the following screen. Also, write the buffer contents to the file so your additions are saved.

```
$ ed try-me<CR>
110
.<CR>
This is the fourth line.
a<CR>
five<CR>
six<CR>
seven<CR>
eight<CR>
nine<CR>
ten<CR>
.<CR>
w<CR>
140
```

Now try adding and subtracting line numbers from the current line.

```
4<CR>
This is the fourth line.
+3<CR>
seven
-5<CR>
This is a second line,
```

The following shows what happens if you ask for a line address that is greater than the last line, or if you try to subtract a number greater than the current line number:

```
5<CR>
five
-6<CR>
?
.=<CR>
5
+7<CR>
?
```

Notice that the current line remains at line 5 of the buffer. The current line changes only if you give **ed** a correct address. The **?** response means there is an error. *Additional Commands and Concepts*, at the end of this chapter, explains how to get a help message that describes the error.

Character String Addresses

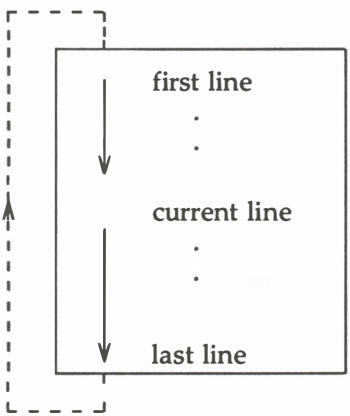
You can search forward or backward in the file for a line containing a particular character string. To do so, specify a string, preceded by a delimiter.

Delimiters mark the boundaries of character strings; they tell **ed** where a string starts and ends. The most common delimiter is slash, used in the following format:

/pattern

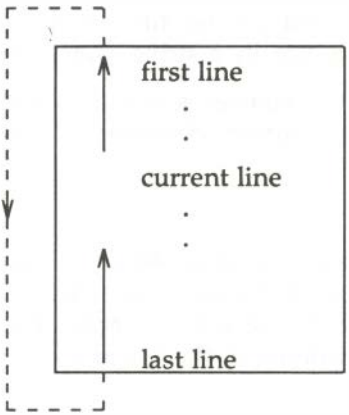
When you specify a pattern preceded by a slash, **ed** begins at the current line and searches forward (down through subsequent lines in the buffer) for the next line containing the *pattern*. When the search reaches the last line of the buffer, **ed** wraps around to the beginning of the file and continues its search from line 1.

The following rectangle represents the editing buffer. The path of the arrows shows the search initiated by a slash.



Another useful delimiter is **?**. If you specify a pattern preceded by a **?**, (as in **?***pattern*), **ed** begins at the current line and searches backward (up through previous lines in the buffer) for the next line containing the *pattern*. If the search reaches the first line of the file, it wraps around and continues searching upward from the last line of the file.

The following rectangle represents the editing buffer. The path of the arrows shows the search initiated by a **?**.



Experiment with these two methods of requesting address searches on the file **try-me**. What happens if **ed** does not find the specified character string?

```
$ ed try-me<CR>
140
.<CR>
ten
?first<CR>
This is the first line of text.
/fourth<CR>
This is the fourth line.
/junk<CR>
?
```

In this example, **ed** found the specified strings **first** and **fourth**. Then, because no command was given with the address, it executed the **p** command by default, displaying the lines it had found. When **ed** cannot find a specified string (e.g., **junk**), it responds with a **?**.

You can also use the slash to search for multiple occurrences of a pattern without typing it more than once. First, specify the pattern by typing */pattern*. After **ed** has printed the first occurrence, it waits for another command. Type slash and press the RETURN key; **ed** continues to search forward through the file for the last *pattern* specified.

Try this command by searching for the word **line** in the file **try-me**:

```
.<CR>
This is the first line of text.
/line<CR>
This is the second line,
/<CR>
and this is the third line.
/<CR>
This is the fourth line.
/<CR>
This is the first line of text.
```

After **ed** has found all occurrences of the *pattern* between the line where you requested a search and the end of the file, it wraps around to the beginning of the file and continues searching.

Specifying a Range of Lines

There are two ways to request a group of lines. You can specify a range of lines, e.g., *address1* through *address2*, or you can specify a global search for all lines containing a specified pattern.

The simplest way to specify a range of lines is to use the line numbers of the first and last lines of the range, separated by a comma. Place this address before the command. For example, if you want to display lines 2 through 7 of the editing buffer, give *address1* as 2 and *address2* as 7 in the following format:

2,7p<CR>

Try this on the file **try-me**:

```
2,7p<CR>
This is the second line,
and this is the third line.
This is the fourth line.
five
six
seven
```

Did you try typing **2,7** without the **p**? What happened? If you do not add the **p** command, **ed** prints only *address2*, the last line of the range of addresses.

Relative line addresses can also be used to request a range of lines. Be sure that *address1* precedes *address2* in the buffer. Relative addresses are calculated from the current line, as the following example shows:

```
4<CR>
This is the fourth line
-2,+3p<CR>
This is the second line,
and this is the third line.
This is the fourth line.
five
six
seven
```

Specifying a Global Search

There are two commands that do not follow the general format of **ed** commands: **g** and **v**. These are global search commands that specify addresses with a character string (*pattern*). The **g** command searches for all lines containing the string *pattern* and performs the *command* on those lines. The **v** command searches for all lines that do not contain the *pattern* and performs the *command* on those lines.

The following is the general format for these commands:

g/*pattern*/*command*<**CR**>
v/*pattern*/*command*<**CR**>

Try these commands by using them to search for the word **line** in **try-me**:

```
g/line/p<CR>
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line
```

```
v/line/p<CR>
five
six
seven
eight
nine
ten
```

Notice the function of the **v** command: it finds all lines that do not contain the word specified in the command line (**line**).

Once again, the default command for the lines addressed by **g** or **v** is **p**; you do not need to include a **p** as the last delimiter on your command line.

```
g/line<CR>
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line
```

However, if you are giving line addresses to be used by other **ed** commands, you need to include beginning and ending delimiters. You can use any of the methods discussed in this section to specify line addresses for **ed** commands. Table 5-2 summarizes the symbols and commands available for addressing lines.

Table 5-2. Summary of Line Addressing

Address	Description
n...	specifies the number of a line in the buffer.
.	specifies the current line (the line most recently acted on by an **ed** command).
.=	asks for the line number of the current line.
$	specifies the last line of the file.
,	indicates the set of lines from line 1 through the last line.
;	indicates the set of lines from the current line through the last line.
+ *n*	indicates the line that is located *n* lines after the current line.
− *n*	indicates the line that is located *n* lines before the current line.
/*abc*	searches forward in the buffer for the first line that contains the pattern *abc*.
?*abc*	searches backward in the buffer for the first line that contains the pattern *abc*.
g/*abc*	indicates the set of all lines that contain the pattern *abc*.
v/*abc*	indicates the set of all lines that do not contain the pattern *abc*.

Exercise 2

2-1. Create a file called **towns** with the following lines:

My kind of town is
Chicago
Like being no where at all in
Toledo
I lost those little town blues in
New York
I lost my heart in
San Francisco
I lost $$ in
Las Vegas

2-2. Display line 3.

2-3. If you specify a range of lines with the relative address −2,+3p, what lines are displayed ?

2-4. What is the current line number? Display the current line.

2-5. What does the last line say?

2-6. What line is displayed by the following request for a search?

?town<CR>

After **ed** responds, type this command alone on a line:

?<CR>

What happened?

2-7. Search for all lines that contain the pattern **in**. Then search for all lines that do not contain the pattern **in**.

Displaying Text in a File

The **ed** text editor provides two commands for displaying lines of text in the editing buffer: **p** and **n**. One displays text with their line numbers; the other displays text without them.

Displaying Text Alone: p Command

You have already used the **p** command in several examples and should be familiar with its general format:

[*address1*[,*address2*]]**p**<**CR**>

p does not take arguments. However, it can be combined with a substitution command line. This is discussed later in this chapter.

Using the line addresses shown in Table 5-3, experiment on a file in your home directory. Try the **p** command with each address and see if **ed** responds as described in the following table.

Table 5-3. Sample Addresses for Displaying Text

Specify this address	Check for this response
1,$p<CR>	**ed** displays the entire file on your terminal.
−5p<CR>	**ed** moves backward five lines from the current line and displays the line found there.
+2p<CR>	**ed** moves forward two lines from the current line and displays the line found there.
1,/*x*/p<CR>	**ed** displays the set of lines from line one through the first line after the current line that contains the character *x*. It is important to enclose the letter *x* within slashes so that **ed** can distinguish between the search pattern address (*x*) and the **ed** command (**p**).

Displaying with Line Numbers: n Command

As the **n** command displays text, it precedes each line with its line number. This is helpful when you are deleting, creating, or changing lines. The general command line format for **n** is the same as that for **p**:

[*address1*,*address2*]**n<CR>**

Like **p**, **n** does not take arguments, but it can be combined with the substitute command.

Try running **n** on the **try-me** file:

```
$ ed try-me<CR>
140
1,$n<CR>
1       This is the first line of text.
2       This is the second line,
3       and this is the third line.
4       This is the fourth line.
5       five
6       six
7       seven
8       eight
9       nine
10      ten
```

Table 5-4 summarizes the **ed** commands for displaying text.

Table 5-4. Summary of Commands for Displaying Text

Command	Function
p	displays on your terminal the specified lines of text in the editing buffer.
n	displays on your terminal the specified lines of text in the editing buffer preceded by their line numbers.

Creating Text

ed has three basic commands for creating new lines of text:

- **a** append text
- **i** insert text
- **c** change text

Appending Text: a Command

The append command, **a**, allows you to add text after the current line or a specified address in the file. You already used this command in the *Getting Started* section of this chapter. The general format for the append command line is:

[*address1*]**a**<**CR**>

Specifying an address is optional. The default value of *address1* is the current line.

In previous exercises, you used this command with the default address. Now try using different line numbers for *address1*. In the following example, a new file called **new-file** is created. In the first append command line, the default address is the current line. In the second append command line, line 1 is specified as *address1*. The lines are displayed with **n** so that you can see their numerical line addresses. Remember, the append mode is ended by typing a period on a line by itself.

```
$ ed new-file<CR>
?new-file
a<CR>
Create some lines
of text in
this file.
.<CR>
1,$n<CR>
1       Create some lines
2       of text in
3       this file.
1a<CR>
This will be line 2<CR>
This will be line 3<CR>
.<CR>
1,$n<CR>
1       Create some lines
2       This will be line 2
3       This will be line 3
4       of text in
5       this file.
```

After you append the two new lines, the line that was originally line 2 (`of text in`) becomes line 4.

You can take shortcuts to places in the file where you want to append text by combining the append command with symbolic addresses. The following three command lines allow you to move through and add to the text quickly in this way:

.a<CR>
appends text after the current line.

$a<CR>
appends text after the last line of the file.

0a<CR>
appends text before the first line of the file (at a symbolic address called line 0).

To try using these addresses, create a one-line file called **lines** and type the examples shown in the following screens. (The examples appear in separate screens for easy reference only; it is not necessary to access the **lines** file three times to try each append symbol. You can access **lines** once and try all three consecutively.)

```
$ ed lines<CR>
?lines
a<CR>
This is the current line.<CR>
.<CR>
p<CR>
This is the current line.
.a<CR>
This line is after the current line.<CR>
.<CR>
-1,.p<CR>
This is the current line.
This line is after the current line.
```

```
$a<CR>
This is the last line now.<CR>
.<CR>
$<CR>
This is the last line now.
```

```
0a<CR>
This is the first line now.<CR>
This is the second line now.<CR>
The line numbers change<CR>
as lines are added.<CR>
.<CR>
1,4n<CR>
1       This is the first line now.
2       This is the second line now.
3       The line numbers change
4       as lines are added.
```

Because the append command creates text after a specified address, the last example refers to the line before line 1 as the line after line 0. To avoid such circuitous references, use another command provided by the editor: the insert command, **i**.

Inserting Text: i Command

The insert command (**i**), allows you to add text before a specified line in the editing buffer. The general command line format for **i** is the same as **a**:

[*address1*]**i**<**CR**>

As with the append command, you can insert one or more lines of text. To quit input mode, you must type a period alone on a line.

Create a file called **insert** in which you can try the insert command (**i**):


```
$ ed insert<CR>
?insert
a<CR>
Line 1<CR>
Line 2<CR>
Line 3<CR>
Line 4<CR>
.<CR>
w<CR>
69
```

Now insert one line of text above line 2 and another above line 1. Use the **n** command to display all lines in the buffer:

```
2i<CR>
This is the new line 2.<CR>
.<CR>
1,$n<CR>
1       Line 1
2       This is the new line 2.
3       Line 2
4       Line 3
5       Line 4
1i<CR>
This is the beginning.<CR>
.<CR>
1,$n<CR>
1       In the beginning
2       Line 1
3       Now this is line 2
4       Line 2
5       Line 3
6       Line 4
```

Experiment with the insert command by combining it with symbolic line addresses:

.i<CR>

$i<CR>

Changing Text: c Command

The change text command (**c**) erases all specified lines and allows you to create one or more lines of text in their place. Because **c** can erase a range of lines, the general format for the command line includes two addresses:

[*address1*,*address2*]**c<CR>**

The change command puts you in text input mode. To leave input mode, type a period alone on a line.

Address1 is the first and *address2* is the last of the range of lines to be replaced by new text. To erase one line of text, specify only *address1*. If no address is specified, **ed** assumes the current line is the line to be changed.

Now create a file called **change** in which you can try this command. After entering the text shown in the screen, change lines one through four by typing **1,4c**.

```
1,5n<CR>
1       line 1
2       line 2
3       line 3
4       line 4
5       line 5
1,4c<CR>
Change line 1<CR>
and lines 2 through 4<CR>
.<CR>
1,$n<CR>
1       change line 1
2       and lines 2 through 4
3       line 5
```

Now experiment with **c** and try to change the current line:

```
.<CR>
line 5
c<CR>
This is the new line 5.
.<CR>
.<CR>
This is the new line 5.
```

If you are not sure you have left text input mode, it is a good idea to type another period. If the current line is displayed, you know you are in the command mode of **ed**.

Table 5-5 summarizes the **ed** commands for creating text.

Table 5-5. Summary of Commands for Creating Text

Command	Function
a	appends text after the specified line in the buffer.
i	inserts text before the specified line in the buffer.
c	changes the text on the specified lines to new text.
.	quits text input mode and return to **ed** command mode.

Exercise 3

3-1. Create a new file called **ex3**. Instead of using the append command to create new text in the empty buffer, try the insert command. What happens?

3-2. Enter **ed** with the file **towns**. What is the current line?

Insert above the third line:

Illinois<CR>

Insert above the current line:

or<CR>
Naperville<CR>

Insert before the last line:

hotels in<CR>

Display the text in the buffer preceded by line numbers.

3-3. In the file **towns**, display lines 1 through 5 and replace lines 2 through 5 with:

London<CR>

Display lines 1 through 3.

3-4. After you complete exercise 3-3, what is the current line?

Find the line of text containing:

```
Toledo
```

Replace

```
Toledo
```

with

Peoria

Display the current line.

3-5. With one command line search for and replace:

```
New York
```

with:

Iron City

Deleting Text

This section discusses two types of commands for deleting text in **ed**. One type is used when you are working in command mode: **d** deletes a line and **u** undoes the last command. The other type of command is used in test input mode: <**CERASE**> deletes a character and <**CKILL**> kills a line. The delete keys that are used in input mode are the same keys you use to delete text that you enter after a shell prompt. They are described in detail in *Correcting Typing Errors* in Chapter 2.

Deleting Lines: d Command

You have already deleted lines of text with the delete command (**d**) in the *Getting Started* section of this chapter.

The general format for the **d** command line is:

[*address1*,*address2*]**d**<**CR**>

You can delete a range of lines (*address1* through *address2*) or you can delete one line only (*address1*). If no address is specified, **ed** deletes the current line.

The next example displays lines one through five and then deletes lines two through four:

```
1,5n<CR>
1           1 horse
2           2 chickens
3           3 ham tacos
4           4 cans of mustard
5           5 bails of hay
2,4d<CR>
1,$n<CR>
1           1 horse
2           5 bails of hay
```

How can you delete only the last line of a file? Using a symbolic line address makes this easy:

$d<CR>

How can you delete the current line? One of the most common errors in **ed** is forgetting to type a period to leave text input mode. When this happens, unwanted text may be added to the buffer. In the next example, a line containing a print command (**1,$p**) is accidentally added to the text before the user leaves input mode. Because this line was the last one added to the text, it becomes the current line. The symbolic address period is used to delete it.

```
a<CR>
Last line of text<CR>
1,$p<CR>
.<CR>
p<CR>
1,$p
.d<CR>
p<CR>
Last line of text.
```

In connection with the delete command, you may also want to learn about the undo command, **u**.

Undoing the Previous Command: u Command

The command **u** (short for undo) nullifies the last command and restores any text changed or deleted by that command. It takes no addresses or arguments. The format is:

u<CR>

One purpose for which the **u** command is useful, is to restore text you have mistakenly deleted. If you delete all lines in a file, then type **p, ed** responds with a **?** because there are no more lines in the file. Use the **u** command to restore them.

```
1,$p<CR>
This is the first line.
This is the middle line.
This is the last line.
1,$d<CR>
p<CR>
?
u<CR>
p<CR>
This is the last line.
```

Now experiment with **u**: use it to undo the append command:

```
.<CR>
This is the only line of text
a<CR>
Add this line<CR>
.<CR>
1,$p<CR>
This is the only line of text
Add this line
u<CR>
1,$p<CR>
This is the only line of text
```

NOTE

u cannot be used to undo the write command (**w**) or the quit command (**q**). However, **u** can undo an undo command (**u**).

Deleting While in Text Input Mode

While in text input mode, you can correct the current line of input with the same keys you use to correct a shell command line. By default, there are two keys available to correct text. The <**CKILL**> key kills the current line. The <**CERASE**> key backs up over one character on the current line so you can retype it, thus, erasing the original character. (See *Correcting Typing Errors* in Chapter 2 for details.)

As mentioned in Chapter 2, you can reassign the line-kill and character-erase functions to other keys if you prefer. (See *Modifying Your Login Environment* in Chapter 7 for instructions.) If you reassigned these functions, you must use the keys you chose while working in **ed**; the default keys (<**CKILL**> and <**CERASE**>) will no longer work.

You may want to include <**CKILL**> or <**CERASE**> as a character (e.g., an @ sign or a # sign) of text. To avoid having these characters interpreted as delete commands, you must precede them with a backslash as shown in the following example:

```
a<CR>
leave San Francisco \@ 20:15 on flight \#347 <CR>
.<CR>
p<CR>
leave San Francisco @ 20:15 on flight #347
```

Table 5-6 summarizes the **ed** commands and shell commands for deleting text in **ed**.

Table 5-6. Summary of Commands for Deleting Text

Command	Function
In command mode:	
<d>	deletes one or more lines of text.
<u>	undoes the previous command.
<CKILL>	deletes the current command line.
In text input mode:	
<CKILL>	deletes the current line.
<CERASE>	deletes the last character typed in.

Substituting Text

You can change your text with a substitute command. This command replaces the first occurrence of a string of characters with new text. The general command line format is:

[*address1*,*address2*]**s**/*old_text*/*new_text*/[*command*]**<CR>**

The following describes each component of the command line:

address1,*address2*
specifies the range of lines being addressed by **s**. The address can be one line, (*address1*), a range of lines (*address1* through *address2*), or a global search address. If no address is given, **ed** makes the substitution on the current line.

s
indicates the substitute command.

/old_text
specifies the text to be replaced. It is usually delimited by slashes, but can be delimited by other characters, e.g., a **?** or a period. The command replaces the first occurrence of these characters that it finds in the text.

/new_text specifies the text to replace *old_text*. It is delimited by slashes or the same delimiters used to specify the *old_text*. It consists of the words or characters that are to replace the *old_text*.

/command
is an option and can be any one of the following four commands:

g
changes all occurrences of *old_text* on the specified lines.

l
displays the last line of substituted text, including nonprinting characters. (See the last section of this chapter, *Additional Commands and Concepts*.)

n
displays the last line of the substituted text preceded by its numerical line address.

p
displays the last line of substituted text.

Substituting on the Current Line

The simplest example of the substitute command is making a change to the current line. You do not need to give a line address for the current line.

s/*old_text*/*new_text*/<**CR**>

The next example contains a typing error. While the line that contains the error is still the current line, you make a substitution to correct it. The old text is the **ai** of **airor** and the new text is **er**.

```
a<CR>
In the beginning, I made an airor.
.<CR>
.p<CR>
In the beginning, I made an airor.
s/ai/er<CR>
```

ed gives no response to the substitute command. To verify that the command has succeeded, you either have to display the line with **p** or **n**, or include **p** or **n** as part of the substitute command line. In the following example, **n** is used to verify that the word **file** has been substituted for the word **toad**:

```
.p<CR>
This is a test toad
s/toad/file/n<CR>
1         This is a test file
```

However, **ed** allows you one shortcut: it prints the results of the command automatically, if you omit the last delimiter after the *new_text* argument.

```
.p<CR>
This is a test file
s/file/frog<CR>
This is a test frog
```

Substituting on One Line

To substitute text on a line that is not the current line, include an address in the command line:

[*address1*]**s**/*old_text*/*new_text*/<**CR**>

For example, in the following screen the command line includes an address for the line to be changed (line 1) because the current line is line 3:

```
1,3p<CR>
This is a pest toad
testing testing
come in toad
.<CR>
come in toad
1s/pest/test<CR>
This is a test toad
```

ed printed the new line automatically after the change was made because the last delimiter was omitted.

Substituting over a Range of Lines

You can make a substitution over a range of lines by specifying the first address (*address1*) through the last address (*address2*):

[*address1*,*address2*]**s**/*old_text*/*new_text*/<**CR**>

If **ed** does not find the pattern to be replaced on a line, no changes are made to that line. An error message (?) is printed if the pattern is not found on any line in *address1, address2*.

In the following example, all lines in the file are addressed for the substitute command. However, only the lines that contain the string **es** (the *old_text* argument) are changed.

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
1,$s/es/ES/n<CR>
4         tESting 1, 2, 3
```

When you specify a range of lines and include **p** or **n** at the end of the substitute line, only the last line changed is printed.

To display all the lines in which text was changed, use the **n** or **p** command with the address **1,$**.

```
1,$n<CR>
1       This is a tESt toad
2       tESting testing
3       come in toad
4       tESting 1, 2, 3
```

Only the first occurrence of **es** (on line 2) has been changed. To change every occurrence of a pattern, use the **g** command, described in the next section.

Global Substitution

One of the most versatile tools in **ed** is global substitution. By placing the **g** command after the last delimiter on the substitute command line, you can change every occurrence of a pattern on the specified lines. Try changing every occurrence of the string as in the last example. If you are following along, doing the examples as you read, remember you can use **u** to undo the last substitute command.

```
u<CR>
1,$p<CR>
This is a test toad
testing, testing
come in toad
testing 1, 2, 3
1,$s/es/ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Another method is to use a global search pattern as an address instead of the range of lines specified by **1,$**.

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/test/s/es/ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

If the global search pattern is unique and matches the argument *old_text* (text to be replaced), you can use an **ed** shortcut: specify the pattern once as the global search address and do not repeat it as an *old_text* argument. **ed** remembers the pattern from the search address and use it again as the pattern to be replaced.

g/*old_text***/s//***new_text***/g<CR>**

NOTE

Whenever you use this shortcut, be sure to include two slashes (//) after the **s**.

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/es/s//ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Experiment with other search pattern addresses:

/*pattern***<CR>**
?*pattern***<CR>**
v/*pattern***<CR>**

See what they do when combined with the substitute command. In the following example, the **v**/*pattern* search format is used to locate lines that do not contain the pattern **testing**. Then the substitute command (**s**) is used to replace the existing pattern (**in**) with a new pattern (**out**) on those lines.

```
v/testing/s/in/out<CR>
This is a test toad
come out toad
```

The line, **This is a test toad**, was also printed even though no substitution was made on it. When the last delimiter is omitted, all lines found with the search address are printed, regardless of whether substitutions have been made on them.

Now search for lines that do contain the pattern **testing** with the **g** command:

```
g/testing/s//jumping<CR>
jumping testing
jumping 1, 2, 3
```

This command makes substitutions only for the first occurrence of the *pattern* (**testing**) in each line. Once again, the lines display on your terminal because the last delimiter has been omitted.

Exercise 4

4-1. In your file **towns** change `town` to **city** on all lines but the line with `little town` on it.

The file should read:

My kind of city is
London
Like being no where at all in
Peoria
I lost those little town blues in
Iron City
I lost my heart in
San Francisco
I lost $$ in
hotels in
Las Vegas

4-2. Try using **?** as a delimiter. Change the current line:

```
Las Vegas
```

to

Toledo

Because you are changing the whole line, you can also do this by using the change command, **c**.

4-3. Try searching backward in the file for the word:

```
lost
```

Substitute using the **?** as the delimiter:

found

Did it work?

4-4. Search forward in the file for

```
no
```

and substitute the following for it:

```
NO
```

What happens if you try to use **?** as a delimiter?

Experiment with the various command combinations available for addressing a range of lines and doing global searches.

What happens if you try to substitute something for the **$$** ? Try to substitute **Big $** for **$** on line 9 of your file. Type:

```
9s/$/Big $<CR>
```

What happened?

Special Characters

If you try to substitute the **$** sign in the following line, you will find that instead of replacing the **$**, the new text is placed at the end of the line:

```
I lost my $ in Las Vegas
```

The **$** is a special character in **ed** that is symbolic for the end of the line.

ed has several special characters that give you a shorthand for search patterns and substitution patterns. The characters act as wild cards. If you have tried to type in any of these characters, the result was probably different than what you had expected.

The special characters are:

.
matches any one character.

*****
matches zero or more occurrences of the preceding character.

.*
matches zero or more occurrences of any character following the period.

^
matches the beginning of the line.

$
matches the end of the line.

****
takes away the special meaning of the special character that follows.

&
repeats the old text to be replaced in the new text of the replacement pattern.

[...]
matches the first occurrence of a character in the brackets.

[^...]
matches the first occurrence of a character that is not in the brackets.

In the following example, **ed** searches for any three-character sequence ending in the pattern at:

```
1,$p<CR>
rat
cat
turtle
cow
goat
g/.at<CR>
rat
cat
goat
```

The word goat is included because the string oat matches the string .at.

The * (asterisk) represents zero or more occurrences of a specified character in a search or substitute pattern. This can be useful in deleting repeated occurrences of a character that have been inserted by mistake. For example, you hold down the **r** key too long while typing the word broke. You can use the * to delete every unnecessary **r** with one substitution command:

```
p<CR>
brrroke
s/br*/br<CR>
broke
```

The substitution pattern includes the **b** before the first **r**. If the **b** were not included in the search pattern, the * would interpret it, during the search, as a zero occurrence of **r**, make the substitution on it, and quit. (Remember, only the first occurrence of a pattern is changed in a substitution, unless you request a global search with **g**.) The following screen shows how the substitution would be made if you did not specify both the **b** and the **r** before the *:

```
p<CR>
brrroke
s/r*/r<CR>
rbrrroke
```

If you combine the period and the *, the combination will match all characters. With this combination, you can replace all characters in the last part of a line:

```
p<CR>
Toads are slimy, cold creatures
s/are.*/are wonderful and warm<CR>
Toads are wonderful and warm
```

The .* can also replace all characters between two patterns:

```
p<CR>
Toads are slimy, cold creatures
s/are.*cre/are wonderful and warm cre<CR>
Toads are wonderful and warm creatures
```


If you want to insert a word at the beginning of a line, use the ^ (circumflex) for the old text to be substituted. This is very helpful when you want to insert the same pattern in the front of several lines. The next example places the word all at the beginning of each line:

```
1,$p<CR>
creatures great and small
things wise and wonderful
things bright and beautiful
1,$s^/all /<CR>
1,$p<CR>
all creatures great and small
all things wise and wonderful
all things bright and beautiful
```

The $ sign is useful for adding characters at the end of a line or a range of lines:

```
1,$p<CR>
I love
I need
I use
The IRS wants my
1,$s/$/ money.<CR>
1,$p<CR>
I love money.
I need money.
I use money.
The IRS wants my money.
```

In these examples, you must remember to put a space after the word all or before the word money because **ed** adds the specified characters to the very beginning or the very end of the sentence. If you forget to leave a space before the word money your file will look like this:

```
1,$s/$/money/<CR>
1,$p<CR>
I lovemoney
I needmoney
I usemoney
The IRS wants mymoney
```

The **$** sign also provides a handy way to add punctuation to the end of a line:

```
1,$p<CR>
I love money
I need money
I use money
The IRS wants my money
1,$s/$/./<CR>
1,$p/<CR>
I love money.
I need money.
I use money.
The IRS wants my money.
```

Because period is not matching any character (old text), but replacing a character (new text), it does not have a special meaning. To change a period in the middle of a line, you must take away the special meaning of the period in the old text. To do this, simply precede the period with a backslash (\.). This is how you take away the special meaning of some special characters that you want to treat as normal text characters in search or substitute arguments. For example, the following screen shows how to take away the special meaning of the period:

```
p<CR>
Way to go.  Wow!
s/\./!<CR>
Way to go!  Wow!
```

The same method can be used with the backslash character itself. If you want to treat a \ as a normal text character, be sure to precede it with a \. For example, if you want to replace the \ symbol with the word backslash, use the substitute command line shown in the following screen:

```
1,2p<CR>
This chapter explains
how to use the \.
s/\\/backslash<CR>
how to use the backslash.
```

If you want to add text without changing the rest of the line, the & (ampersand) provides a useful shortcut. The & repeats the old text in the replacement pattern, so you do not have to type the pattern twice. For example:

```
p<CR>
The neanderthal skeletal remains
s/thal/& man's/<CR>
p<CR>
The neanderthal man's skeletal remains
```

ed automatically remembers the last string of characters in a search pattern or the old text in a substitution. However, you must prompt **ed** to repeat the replacement characters in a substitution with % (percent sign). % allows you to make the same substitution on multiple lines without requesting a global substitution. For example, to change the word money to the word gold, repeat the last substitution from line 1 on line 3, but not on line 4.


```
1,$n<CR>
 1          I love money
 2          I need food
 3          I use money
 4          The IRS wants my money
 1s/money/gold<CR>
 I love gold
 3s//%<CR>
I use gold
 1,$n<CR>
 1          I love gold
 2          I need food
 3          I use gold
 4          The IRS wants my money
```

ed automatically remembers the word **money** (the old text to be replaced), so that string does not have to be repeated between the first two delimiters. % tells **ed** to use the last replacement pattern, gold.

ed tries to match the first occurrence of one of the characters enclosed in brackets and substitute the specified old text with new text. The brackets can be at any position in the pattern to be replaced.

In the following example, **ed** changes the first occurrence of the numbers **6, 7, 8,** or **9** to **4** on each line in which it finds one of those numbers:

```
1,$p<CR>
Monday      33,000
Tuesday     75,000
Wednesday   88,000
Thursday    62,000
1,$s/[6789]/4<CR>
Monday      33,000
Tuesday     45,000
Wednesday   48,000
Thursday    42,000
```

The next example deletes the Mr or Ms from a list of names:

```
1,$p<CR>
Mr Arthur Middleton
Mr Matt Lewis
Ms Anna Kelley
Ms M. L. Hodel
1,$s/M[rs] //<CR>
1,$p<CR>
Arthur Middleton
Matt Lewis
Anna Kelley
M. L. Hodel
```

If a circumflex is the first character in brackets, **ed** interprets it as an instruction to match characters that are not within the brackets. However, if the circumflex is in any other position within the brackets, **ed** interprets it literally, as a circumflex.

```
1,$p<CR>
grade   A   Computer Science
grade   B   Robot Design
grade   A   Boolean Algebra
grade   D   Jogging
grade   C   Tennis
1,$s/grade [^AB]/gradeA<CR>
1,$p<CR>
grade   A   Computer Science
grade   B   Robot Design
grade   A   Boolean Algebra
grade   A   Jogging
grade   A   Tennis
```

Whenever you use special characters as wild cards in the text to be changed, remember to use a unique pattern of characters. In the above example, if you had used only you would have changed the g in the word grade to A:

1,$s/[^AB]/A<CR>

Experiment with these special characters. Find out what happens (or does not happen) if you use them in different combinations.

Table 5-7 summarizes the special characters for search or substitute patterns.

Table 5-7. Summary of Special Characters

Command	Function
.	matches any one character in a search or substitute pattern.
*	matches zero or more occurrences of the preceding character in a search or substitute pattern.
.*	matches zero or more occurrences of any characters following the period.
^	matches the beginning of the line in the substitute pattern to be replaced or in a search pattern.
$	matches the end of the line in the substitute pattern to be replaced.
\	takes away the special meaning of the special character that follows in the substitute or search pattern.
&	repeats the old text to be replaced in the new text replacement pattern.
%	matches the last replacement pattern.
[...]	matches the first occurrence of a character in the brackets.
[^...]	matches the first occurrence of a character that is not in the brackets.

Exercise 5

5-1. Create a file that contains the following lines of text:

A Computer Science
D Jogging
C Tennis

What happens if you try this command line:

1,$s/[^AB]/A/<CR>

Undo the above command. How can you make the C and D unique? (Hint: they are at the beginning of the line, in the position shown by the ^.) Do not be afraid to experiment!

5-2. Insert the following line above line 2:

These are not really my grades.

Using brackets and the ^ character, create a search pattern that you can use to locate the line you inserted. There are several ways to address a line. When you edit text, use the way that is quickest and easiest for you.

5-3. Add the following lines to your file:

I love money
I need money
The IRS wants my money

Now use one command to change them to:

It's my money
It's my money
The IRS wants my money

Using two command lines, do the following: change the word on the first line from **money** to **gold**, and change the last two lines from **money** to **gold** without using the words **money** or **gold** themselves.

5-4. How can you change the line:

```
1020231020
```

to the following without repeating the old digits in the replacement pattern?

```
10202031020
```

5-5. Create a line of text containing the following characters:

```
* . \ & % ^ *
```

Substitute a letter for each character. Do you need to use a backslash for every substitution?

Moving Text

You have now learned to address lines, create and delete text, and make substitutions. **ed** has one more set of versatile and important commands. You can move, copy, or join lines of text in the editing buffer. You can also read in text from a file that is not in the editing buffer, or write lines of the file in the buffer to another file in the current directory. The commands that move text are:

m
moves lines of text.

t
copies lines of text.

j
joins contiguous lines of text.

w
writes lines of text to a file.

r
reads in the contents of a file.

Moving Lines of Text

The **m** command allows you to move blocks of text to another place in the file. The general format is:

[*address1*,*address2*]**m**[*address3*]<**CR**>

The components of this command line include:

address1,*address2*
specifies the range of lines to be moved. If only one line is moved, only *address1* is given. If no address is given, the current line is moved.

m
indicates the move command.

address3
places the text after this line.

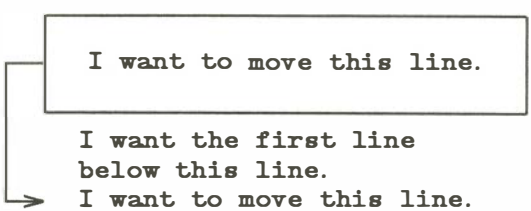
Try the following example to see how the command works. Create a file that contains these three lines of text:

I want to move this line.
I want the first line
below this line.

Type:

1m3<**CR**>

ed will move line 1 below line 3.



The next screen shows how this will appear on your terminal:

```
1,$p<CR>
I want to move this line.
I want the first line
below this line.
1m3<CR>
1,$p<CR>
I want the first line
below this line.
I want to move this line.
```

If you want to move a paragraph of text, have *address1* and *address2* define the range of lines of the paragraph.

In the following example, a block of text (lines 8 through 12) is moved below line 65. Notice the **n** command that prints the line numbers of the file.

```
8,12n<CR>
8       This is line 8.
9       It is the beginning of a
10      very short paragraph.
11      This paragraph ends
12      on this line.
64,65n<CR>
64      Move the block of text
65      below this line.
8,12m65<CR>
59,65n<CR>
59      Move the block of text
60      below this line.
61      This is line 8.
62      It is the beginning of a
63      very short paragraph.
64      This paragraph ends
65      on this line.
```

How can you move lines above the first line of the file? Try the following command:

3,4m0<CR>

When *address3* is 0, the lines are placed at the beginning of the file.

Copying Lines of Text

The copy command **t** (transfer) acts like the **m** command except that the block of text is not deleted at the original address of the line. A copy of that block of text is placed after a specified line of text. The general format of the command line is also similar.

The general format of the **t** command also looks like the **m** command:

[*address1*,*address2*]**t**[*address3*]<**CR**>

address1,*address2*
specifies the range of lines to be copied. If only one line is copied, only *address1* is given. If no address is given, the current line is copied.

t
specifies the copy command.

address3
places the copy of the text after this line.

The next example shows how to copy three lines of text below the last line.

```
                Safety procedures:

        If there is a fire in the building:
        Close the door of the room to seal off the fire

        Break glass of nearest alarm.
        Pull lever.
        Locate and use fire extinguisher.
                .
                .
                .
        A chemical fire in the lab requires that you:

        Break glass of nearest alarm
        Pull lever
        Locate and use fire extinguisher
```

The commands and **ed**s responses to them are displayed in the next screen. Again, the **n** command displays the line numbers.

```
5,8n<CR>
5       Close the door of the room, to seal off the fire.
6       Break glass of nearest alarm.
7       Pull lever.
8       Locate and use fire extinguisher.
30n<CR>
30      A chemical fire in the lab requires that you:
6,8t30<CR>
30,$n<CR>
30      A chemical fire in the lab requires that you:
31      Break glass of nearest alarm
32      Pull lever
33      Locate and use fire extinguisher
6,8n<CR>
6       Break glass of nearest alarm
7       Pull lever
8       Locate and use fire extinguisher
```

The text in lines 6 through 8 remains in place. A copy of those three lines is placed after line 50.

Experiment with **m** and **t** on one of your files.

Joining Contiguous Lines

The **j** command joins the current line with the following line. The general format is:

[*address1*,*address1*]**j**<**CR**>

The next example shows how to join several lines together. An easy way of doing this is to display the lines you want to join using **p** or **n**.

```
1,2p<CR>
Now is the time to join
the team.
p<CR>
the team.
1p<CR>
Now is the time to join
j<CR>
p<CR>
Now is the time to join the team.
```

There is no space between the last word (`join`) and the first word of the next line (`the`), and the last word (`play`). You must place a space between them by using the **s** command.

Writing Lines of Text to a File

The **w** command writes text from the buffer into a file. The general format is:

[*address1*,*address2*]**w** [*filename*]<**CR**>

address1,*address2*
specifies the range of lines to be placed in another file. If you do not use *address1* or *address2*, the entire file is written into a new file.

w
specifies the write command.

filename
specifies the name of the new file that contains a copy of the block of text.

In the following example the body of a letter is saved in a file called **memo**, so that it can be sent to other people:

```
1,$n<CR>
1                March 17, 1986
2          Dear Kelly,
3          There will be a meeting in the
4          green room at 4:30 P.M. today.
5          Refreshments will be served.
3,6w memo<CR>
87
```


The **w** command places a copy of lines three through six into a new file called **memo**. **ed** responds with the number of characters in the new file.

The **w** command overwrites pre-existing files; it erases the current file and puts the new block of text in the file without warning you. If, in our example, a file called **memo** had existed before we wrote our new file to that name, the original file would have been erased.

In *Additional Commands and Concepts*, later in this chapter, you will learn how to execute shell commands from **ed**. Then you can list the file names in the directory to make sure that you are not overwriting a file.

Another potential problem is that you cannot write other lines to the file **memo**. If you try to add lines 13 through 16, the existing lines (3 through 6) will be erased and the file will contain only the new lines (13 through 16).

Reading in the Contents of a File

The **r** command can be used to append text from a file to the buffer. The general format for the read command is:

[*address1*]**r** *filename*<**CR**>

address1
specifies that the text will be placed after the line *address1*. If *address1* is not given, the file is added to the end of the buffer.

r
specifies the read command.

filename
specifies the name of the file that will be copied into the editing buffer.

Using the example from the write command, the next screen shows a file being edited and new text being read into it.

```
1,$n<CR>
1                    March 17, 1986
2          Dear Michael,
3          Are you free later today?
4          Hope to see you there.
3r memo<CR>
87
3,$n<CR>
3          Are you free later today?
4          There is a meeting in the
5          green room at 4:30 P.M. today.
6          Refreshments will be served.
7          Hope to see you there.
```

ed responds to the read command with the number of characters in the file being added to the buffer (in the example, **memo**).

It is a good idea to display new or changed lines of text to be sure that they are correct.

Table 5-8 summarizes the **ed** commands for moving text.

Table 5-8. Summary of **ed** Commands for Moving Text

Command	Function
m	moves lines of text.
t	copies lines of text.
j	joins contiguous lines.
w	writes text into a new file.
r	reads in text from another file.

Exercise 6

6-1. There are two ways to copy lines of text in the buffer: by issuing the copy command, or by using the write and read commands to first write text to a file and then read the file into the buffer.

Writing to a file and then reading the file into the buffer is a longer process. Can you think of an example where this method would be more practical?

What commands can you use to copy lines 10 through 17 of file **exer** into the file **exer6** at line 7?

6-2. Lines 33 through 46 give an example that you want placed after line 3, and not after line 32. What command performs this task?

6-3. If you are on line 10 of a file and you want to join lines 13 and 14. What commands can you issue to do this?

Additional Commands and Concepts

There are four other commands and a special file that is useful during editing sessions.

h,H	accesses the help commands, which provide error messages.
l	displays characters that are not normally displayed.
f	displays the current file name.
!	temporarily escapes **ed** to execute a shell command.
ed.hup	When a system interrupt occurs, the **ed** buffer is saved in a special file named **ed.hup**.

Help Commands

You may have noticed when you were editing a file that **ed** responds to some of your commands with a **?**. The **?** is a diagnostic message issued by **ed** when it has found an error. The help commands give you a short message to explain the reason for the most recent diagnostic.

There are two help commands:

- **h** displays a short error message that explains the most recent **?**.
- **H** places **ed** into help mode so that a short error message displays every time the **?** appears. (To cancel this request, type **H**.)

You know that if you try to quit **ed** without writing the changes in the buffer to a file, you will get a **?**. Do this now. When the **?** appears, type **h**:

```
q<CR>
?
h<CR>
warning: expecting `w'
```

The **?** also displays when you specify a new file name on the **ed** command line. Give **ed** a new file name. When the **?** appears, type **h** to find out what the error message means.

```
ed newfile<CR>
? newfile
h<CR>
cannot open input file
```

This message means one of two things: either there is no file called **newfile** or the file exists but **ed** is not allowed to read it.

As explained earlier, the **H** command responds to the **?** then turns on the help mode of **ed**, so that **ed** gives you a diagnostic explanation every time the **?** displays. To turn off help mode, type **H** again. The next screen shows **H** being used to turn on help mode. Sample error messages also display in response to some common mistakes.

```
$ ed newfile<CR>
e newfile<CR>
?newfile
H<CR>
cannot open input file
/hello<CR>
?
illegal suffix
1,22p<CR>
?
line out of range
a<CR>
I am appending this line to the buffer.
.<CR>
s/$ tea party<CR>
?
illegal or missing delimiter
,$s/$/ tea party<CR>
?
unknown command
H<CR>
q<CR>
?
h<CR>
warning: expecting `w'
```

The following are some common error messages you may encounter during editing sessions:

`illegal suffix`
: **ed** cannot find an occurrence of the search pattern **hello** because the buffer is empty.

`line out of range`
: **ed** cannot print any lines because the buffer is empty or the line specified is not in the buffer.

A line of text is appended to the buffer to show you some error messages associated with the **s** command:

illegal or missing delimiter
The delimiter between the old text to be replaced and the new text is missing.

unknown command
address1 was not typed in before the comma; **ed** does not recognize **,$**.

Help mode is then turned off and **h** is used to determine the meaning of the last **?** . While you are learning **ed**, you may want to leave help mode turned on. If so, use the **H** command. However, once you become adept at using **ed**, you will only need to see error messages occasionally. Then you can use the **h** command.

Displaying Nonprinting Characters

If you are typing a tab character, the terminal normally displays up to eight spaces (covering the space up to the next tab setting). (Your tab setting may be more or less than eight spaces; see Chapter 7, *Shell Tutorial*, on setting **stty**.)

If you want to see how many tabs you have inserted into your text, use the **l** (list) command. The general format for the **l** command is the same as for **n** and **p**.

[*address1*,*address2*]**l**<**CR**>

The components of this command line are:

address1,*address2*
specifies the range of lines to be displayed. If no address is given, the current line displays. If only *address1* is given, only that line displays.

l
is the command that displays the nonprinting characters with the text.

The **l** command denotes tabs with a > (greater than) character. To type control characters, hold down the CTRL key and press the appropriate alphabetic key. The key that sounds the bell is **^g** (CTRL-**g**). It displays as \07, which is the octal representation (the computer's code) for **^g**.

Type in two lines of text that contain a <^g> and a tab. Then use the **l** command to display the lines of text on your terminal.

```
a<CR>
Add a <^g> (CTRL-g) to this line.<CR>
Add a <tab> (tab) to this line.<CR>
.<CR>
1,2l<CR>
Add a \07 (CTRL-g) to this line.<CR>
Add a > (tab) to this line.<CR>
```

Did the bell sound when you typed <^g>?

Checking the Current File Name

In a long editing session, you may forget the file name. The **f** command can tell you which file is currently in the buffer. Or, you may want to preserve the original file that you entered into the editing buffer and write the contents of the buffer to a new file. In a long editing session, you may forget and accidentally overwrite the original file with the customary **w** and **q** command sequence. You can prevent this by telling the editor to associate the contents of the buffer with a new file name while you are in the middle of the editing session. This is done with the **f** command and a new file name.

The format for displaying the current file name is **f** alone on a line:

f<CR>

To see how **f** works, enter **ed** with a file. For example, if your file is called **oldfile, ed** will respond as shown in the following screen:

```
ed oldfile<CR>
323
f<CR>
oldfile
```

To associate the contents of the editing buffer with a new file name use this general format:

f newfile<CR>

If no file name is specified with the write command, **ed** remembers the file name given at the beginning of the editing session and writes to that file. If you do not want to overwrite the original file, you must either use a new file name with the write command, or change the current file name using the **f** command followed by the new file name. Because you can use **f** at any point in an editing session, you can change the file name immediately. You can then continue with the editing session without worrying about overwriting the original file.

The next screen shows the commands for entering the editor with **oldfile**, then changing its name to **newfile**. A line of text is added to the buffer, then the write and quit commands are issued.

```
ed oldfile<CR>
323
f<CR>
oldfile
f newfile<CR>
newfile
a<CR>
Add a line of text.<CR>
.<CR>
w<CR>
343
q<CR>
```

Once you have returned to the shell, you can list your files and verify the existence of the new file, **newfile**. **newfile** should contain a copy of the contents of **oldfile** plus the new line of text.

Executing a Shell Command

How can you make sure you are not overwriting an existing file when you write the contents of the editor to a new file name? You need to return to the shell to list your files. The **!** allows you to temporarily return to the shell, execute a shell command, and then return to the current line of the editor.

The general format for the escape sequence is:

```
!shell command line<CR>
shell response to the command line
!
```

When you type the **!** as the first character on a line, the shell command must follow on that same line. The programs response to your command appears as the command is running. When the command has finished executing, the **!** appears alone on a line. This means that that you are back in the editor at the current line.

For example, if you want to return to the shell to find out the correct date, type **!** and the shell command **date**.

```
p<CR>
This is the current line
! date<CR>
Tue  Apr 1  14:24:22  EST  1986
!
p<CR>
This is the current line.
```

The screen first displays the current line. Then the command is given to temporarily leave the editor and display the date. After the date displays, you are returned to the current line of the editor.

If you want to execute more than one command on the shell command line, see the discussion on **;** in the section called *Special Characters* in Chapter 7.

Recovering from System Interrupts

What happens if you are creating text in **ed** and there is an interrupt to the system, you are accidentally hung up on the system, or your terminal is unplugged? When an interrupt occurs, the operating system tries to save the contents of the editing buffer in a special file named **ed.hup**. Later, you can retrieve your text from this file in one of two ways. First, you can use a shell command to move **ed.hup** to another file name, e.g., the name the file had while you were editing it (before the interrupt). Second, you can enter **ed** and use the **f** command to rename the contents of the buffer.

An example of the second method is shown in the following screen:

```
ed ed.hup<CR>
928
f myfile<CR>
myfile
```

If you use the second method to recover the contents of the buffer, be sure to remove the **ed.hup** file afterward.

Conclusion

You are now familiar with many useful commands in **ed**. The commands that were not discussed in this tutorial, such as **G**, **P**, **Q** and the use of **( )** and { }, are discussed on the **ed**(1) page of the *User's Reference Manual*. You can experiment with these commands and try them to see what tasks they perform.

Table 5-9 summarizes the commands covered in this section.

Table 5-9. Summary of Additional Commands and Concepts

Command	Function
h	displays a short error message for the preceding diagnostic **?**.
H	turns on help mode. An error message will be given with each diagnostic **?**. The second **H** turns off help mode.
l	displays nonprinting characters in the text.
f	displays the current file name.
f *newfile*	changes the current file name associated with the editing buffer to *newfile*.
!cmd	temporarily escapes to the shell to execute the specified shell command *cmd*.
ed.hup	The editing buffer is saved in **ed.hup** if the terminal is hung up before a write command.

Exercise 7

7-1. Create a new file called **newfile1**. Access **ed** and change the files name to **current1**. Then create some text and write and quit **ed**. Run the **ls** command to verify that there is not a file called **newfile1** in your directory. If you do the shell command **ls**, you will see the directory does not contain a file called **newfile1**.

7-2. Create a file named **file1**. Append some lines of text to the file. Leave append mode but do not write the file. Turn off your terminal. Then turn on your terminal and log in again. Issue the **ls** command in the shell. Is there a new file called **ed.hup**? Place **ed.hup** in **ed**. How can you change the current file name to **file1**? Display the contents of the file. Are the lines the same lines you created before you turned off your terminal?

7-3. While you are in **ed**, temporarily escape to the shell and send a mail message to yourself.

Answers to Exercises

Exercise 1

1-1.

```
$ ed junk<CR>
? junk
a<CR>
Hello world.<CR>
.<CR>
w<CR>
12
q<CR>
$
```

1-2.

```
$ ed junk<CR>
12
1,$p<CR>
Hello world.<CR>
q<CR>
$
```

The system did not respond with the warning question mark because you did not make any changes to the buffer.

1-3.

```
$ ed junk<CR>
12
a<CR>
Wendy's horse came through the window.<CR>
.<CR>
1,$p<CR>
Hello world.
Wendy's horse came through the window.
q<CR>
?
w stuff<CR>
60
q<CR>
$
```

Exercise 2

2-1.

```
$ ed towns<CR>
? towns
a<CR>
My kind of town is<CR>
Chicago<CR>
Like being no where at all in<CR>
Toledo<CR>
I lost those little town blues in<CR>
New York<CR>
I lost my heart in<CR>
San Francisco<CR>
I lost $$ in<CR>
Las Vegas<CR>
.<CR>
w<CR>
164
```

2-2.

```
3<CR>
Like being no where at all in
```

2-3.

```
-2,+3p<CR>
My kind of town is
Chicago
Like being no where at all in
Toledo
I lost those little town blues in
New York
```

2-4.

```
.=<CR>
6
6<CR>
New York
```

2-5.

```
$<CR>
Las Vegas
```

2-6.

```
?town<CR>
I lost those little town blues in
?<CR>
My kind of town is
```

2-7.

```
g/in<CR>
My kind of town is
Like being no where at all in
I lost those little town blues in
I lost my heart in
I lost $$ in

v/in<CR>
Chicago
Toledo
New York
San Francisco
Las Vegas
```

Exercise 3

3-1.

```
$ ed ex3<CR>
?ex3
i<CR>
?
q<CR>
```

The **?** after the **i** means there is an error in the command. There is no current line before which text can be inserted.

3-2.

```
$ ed towns<CR>
164
.n<CR>
10          Las Vegas
3i<CR>
Illinois<CR>
.<CR>
.i<CR>
or<CR>
Naperville<CR>
.<CR>
$i<CR>
hotels in<CR>
1,$n<CR>
 1  my kind of town is
 2  Chicago
 3  or
 4  Naperville
 5  Illinois
 6  Like being no where at all in
 7  Toledo
 8  I lost those little town blues in
 9  New York
10  I lost my heart in
11  San Francisco
12  I lost $$ in
13  hotels in
14  Las Vegas
```

3-3.

```
1,5n<CR>
1  My kind of town is
2  Chicago
3  or
4  Naperville
5  Illinois
2,5c<CR>
London<CR>
.<CR>
1,3n<CR>
1  My kind of town is
2  London
3  Like being no where at all
```

3-4.

```
.<CR>
Like being no where at all
/Tol<CR>
Toledo
c<CR>
Peoria<CR>
.<CR>
.<CR>
Peoria
```

3-5.

```
.<CR>
/New Y/c<CR>
Iron City<CR>
.<CR>
.<CR>
Iron City
```

Your search string need not be the entire word or line. It only needs to be unique.

Exercise 4

4-1.

```
v/little town/s/town/city<CR>
My kind of city is
London
Like being no where at all in
Peoria
Iron City
I lost my heart in
San Francisco
I lost $$ in
hotels in
Las Vegas
```

The following line was not printed because it wat not addressed by the **v** command.

```
I lost those little town blues in
```

4-2.

```
.<CR>
Las Vegas
s?Las Vegas?Toledo<CR>
Toledo
```

4-3.

```
?lost?s??found<CR>
I found $$ in
```

4-4.

```
/no?s??NO<CR>
?
/no/s//NO<CR>
Like being NO where at all in
```

You cannot mix delimiters such as **/** and **?** in a command line.

The substitution command on line 9 produced this output:

```
I found $$ inBig $
```

It did not work correctly because the $ sign is a special character in **ed**.

Exercise 5

5-1.

```
$ ed file1<CR>
? file1
a<CR>
A  Computer Science<CR>
D  Jogging<CR>
C  Tennis<CR>
.<CR>
1,$s/[^AB]/A/<CR>
1,$p<CR>
A   Computer Science
A   Jogging
A   Tennis
u<CR>
```

```
1,$s/^[^AB]/A<CR>
1,$p<CR>
A   Computer Science
A   Jogging
A   Tennis
```

5-2.

```
2i<CR>
These are not really my grades.<CR>
1,$p<CR>
A  Computer Science
These are not really my grades.
A  Tennis
A  Jogging
/^[^A]<CR>
These are not really my grades
?^[T]<CR>
These are not really my grades
```

5-3.

```
1,$p<CR>
I love money
I need money
The IRS wants my money
g/^I/s/I.*m /It's my m<CR>
It's my money
It's my money
```

```
/s/money/gold<CR>
It's my gold
2,$s//%<CR>
The IRS wants my gold
```

5-4.

```
s/10202/&0<CR>
10202031020
```

5-5.

```
a<CR>
* . \ & % ^ *<CR>
.<CR>
s/*/a<CR>
a . \ & % ^ *
s/*/b<CR>
a . \ & % ^ b
```

Because there were no preceding characters, * substituted for itself.

```
s/\c<CR>
a c \ & % ^ b
s/\\d<CR>
a c d & % ^ b
s/&/e<CR>
a c d e % ^ b
s/%/f<CR>
a c d e f ^ b
```

The **&** and **%** are only special characters in the replacement text.

```
s/\'/g<CR>
a c d e f g b
```

Exercise 6

6-1. Any time you have lines of text that you may want to have repeated several times, it may be easier to write those lines to a file and read in the file at those points in the text.

If you want to copy the lines into another file you must write them to a file, then read that file into the buffer containing the other file.

```
ed exer<CR>
725
10,17 w temp<CR>
210
q<CR>
ed exer6<CR>
305
7r temp<CR>
210
```

The file **temp** can be called any file name.

6-2.

```
33,46m3<CR>
```

6-3.

```
.=<CR>
10
13p<CR>
This is line 13.
j<CR>
.p<CR>
This is line 13.and line 14.
```

Remember that .= gives you the current line.

Exercise 7

7-1.

```
$ ed newfile1<CR>
? newfile1
f current1<CR>
current1
a<CR>
This is a line of text<CR>
Will it go into newfile1<CR>
or into current1<CR>
.<CR>
w<CR>
66
q<CR>
$ ls<CR>
bin
current1
```

7-2.

```
ed file1<CR>
? file1
a<CR>
I am adding text to this file.<CR>
Will it show up in ed.hup?<CR>
.<CR>
```

Turn off your terminal.

Log in again.

```
ed ed.hup<CR>
58
f file1<CR>
file1
1,$p<CR>
I am adding text to this file.
Will it show up in ed.hup?
```

7-3.

```
$ ed file1<CR>
58
! mail mylogin<CR>
You will get mail when<CR>
you are done editing!<CR>
.<CR>
!
```

6 Screen Editor Tutorial (vi)

Introduction

This chapter is a tutorial on the screen editor, **vi** (short for visual editor). The **vi** editor is a powerful and sophisticated tool for creating and editing files. It is designed for use with a video display terminal which is used as a window through which you can view the text of a file. A few simple commands allow you to make changes to the text that are quickly reflected on the screen.

The **vi** editor displays from one to many lines of text. It allows you to move the cursor to any point on the screen or in the file (by specifying places such as the beginning or end of a word, line, sentence, paragraph, or file) and create, change, or delete text from that point. You can also use some line editor commands, such as the powerful global commands that allow you to change multiple occurrences of the same character string by issuing one command. To move through the file, you can scroll the text forward or backward, revealing the lines below or above the current window, as shown in Figure 6-1.

NOTE

Not all terminals have text scrolling capability; whether or not you can take advantage of **vi**s scrolling feature depends on the type of terminal you have.

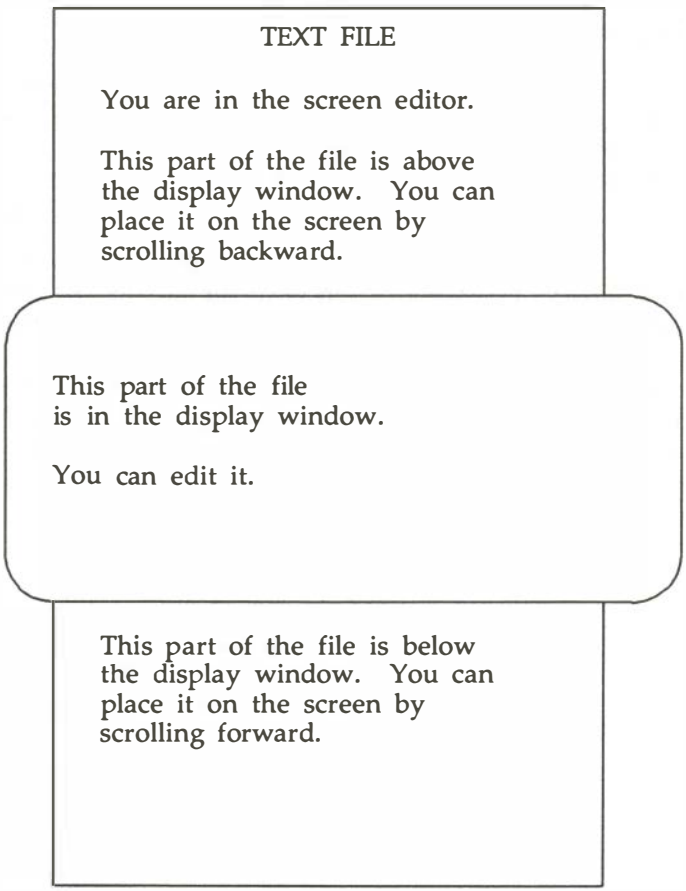


Figure 6-1. Displaying a File with a **vi** Window

There are more than 100 commands within **vi**. This chapter covers the basic commands that will enable you to use **vi** simply but effectively. Specifically, it explains how to do the following tasks:

- Set up your terminal so that **vi** is accessible.
- Enter **vi**, create text, delete mistakes, write the text to a file, and quit.
- Move text within a file.
- Electronically cut and paste text.
- Use special commands and shortcuts.
- Temporarily escape to the shell to execute shell commands.
- Use line editing commands available within **vi**.
- Edit several files in the same session.
- Recover a file lost by an interruption to an editing session.
- Change your shell environment to set your terminal configuration and an automatic carriage return.

As you read this tutorial, keep in mind the notations and conventions described in Chapter 1, *What is the SYSTEM V/88 System?* Note that the arrows in the screen illustrations are used to show the position of the cursor.

The set of commands discussed in each section are summarized at the end of the section and are listed by topic in the summary of **vi** commands in Appendix D. At the end of some sections, exercises are given so you can experiment. Their answers are collected at the end of this chapter. The best way to learn **vi** is by doing the examples and exercises as you read the tutorial. Log in on the operating system when you are ready to read this chapter.

Getting Started

The operating system is flexible; it can be accessed from many kinds of terminals. However, because it is internally structured to be able to operate in so many ways, it needs to know what kind of hardware is being used.

In addition, the operating system offers various optional features for using your terminal that you may or may not want to incorporate into your computing session routine. Your choice of these options, together with your hardware specifications, comprise your login environment. Once you have set up your login environment, the shell implements these specifications and options automatically every time you log in.

This section describes two parts of the login environment: setting the terminal configuration, which is essential for using **vi** properly, and setting the wrap margin, or automatic (carriage) RETURN.

Setting the Terminal Configuration

Before you enter **vi**, you must set your terminal configuration. This simply means that you tell the operating system what type of terminal you are using. This is necessary because the software for the **vi** editor is executed differently on different terminals.

Each type of terminal has several code names that are recognized by the operating system. Appendix F, *Acceptable Terminal Names* section, tells you how to find a recognized name for your terminal. Keep in mind that many computer installations add terminal types to the list of terminals supported by default in your operating system. It is a good idea to check with your local System Administrator for the most up-to-date list of available terminal types.

To see what your current terminal configuration is, type

echo TERM

To reset your terminal configuration, type

TERM=*terminal_name*<**CR**>
export TERM<**CR**>
TermSetup init<**CR**>

The first line puts a value (a terminal type) in a variable called TERM. The second line exports this value; it conveys the value to all operating system programs whose execution depends on the type of terminal being used.

The **TermSetup** command on the third line initializes (sets up) the software in your terminal so that it functions properly with the operating system. It is essential to run the **TermSetup** command when you are setting your terminal configuration because terminal functions, e.g., tab settings, will not work properly unless you do. This command executes the **tput**(1) commands necessary for the terminal.

For example, if your terminal is a vt100, this is how your commands appear on the screen:

```
$ TERM=vt100<CR>
$ export TERM<CR>
$  TermSetup<CR>
```

Do not experiment by entering names for terminal types other than your terminal. This might confuse the operating system, and you may have to log off, hang up, or get help from your System Administrator to restore your login environment.

Changing Your Environment

If you are going to use **vi** regularly, you should change your login environment permanently so you do not have to configure your terminal each time you log in. Your login environment is controlled by a file in your home directory called **.profile**. (This file, pronounced dot profile, is created automatically by the **sysadm adduser** command, by copying **/etc/stdprofile**. For details, see Chapter 7 and Appendix F.)

If you specify the setting for your terminal configuration in your **.profile**, your terminal is configured automatically every time you log in. You can do this by adding the three lines shown in the last screen (the **TERM** assignment, **export** command, and **TermSetup** command) to your **.profile**. (For detailed instructions, see Chapter 7 and Appendix F.) If your **.profile** is modeled after, or copied from, **/etc/stdprofile**, you need only set **TERM**=*terminal name*. Even this is not necessary if the file **local/bin/TermAssume** is set up correctly.

Setting the Automatic RETURN

NOTE

To set an automatic RETURN, you must know how to create a file. If you are familiar with another text editor, such as **ed**, follow the instructions in this section. If you do not know how to use an editor but would like to have an automatic RETURN setting, skip this section for now and return to it when you have learned the basic skills taught in this chapter.

If you want the RETURN key to be entered automatically, create a file called **.exrc** in your home directory. You can use the **.exrc** file to contain options that control the **vi** editing environment.

To create a **.exrc** file, enter an editor, giving it that file name. Then type in one line of text: a specification for the wrapmargin (automatic carriage return) option. The format for this option specification is:

wm=*n***<CR>**

n represents the number of characters from the righthand side of the screen where you want an automatic carriage return to occur. For example, you want a carriage return at 20 characters from the righthand side of the screen:

wm=20<CR>

Finally, write the buffer contents to the file and quit the editor (see *Text Editing Buffers* in Chapter 4). The next time you log in, this file will give you an automatic RETURN.

To check your settings for the terminal and wrapmargin when you are in **vi**, enter the command:

:set<CR>

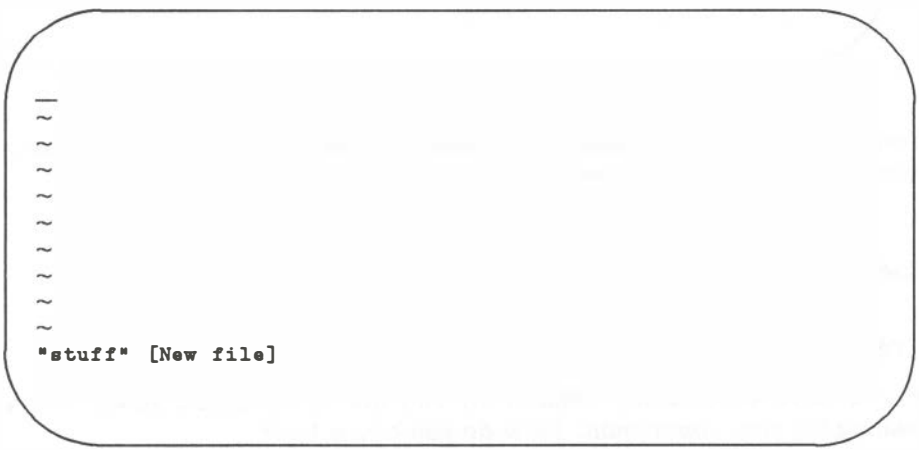
vi reports the terminal type and the wrapmargin, as well as any other options you may have specified. You can also use the **:set** command to create or change the wrapmargin option. Try experimenting with it.

Creating a File

First, enter the editor: type **vi** and the name of the file you want to create or edit:

vi *filename*<**CR**>

For example, you want to create a file called **stuff**. When you type the **vi** command with the file name **stuff**, **vi** clears the screen and displays a window in which you can enter and edit text:



The _ (underscore) on the top line represents the cursor waiting for you to enter a command there. (On video display terminals, the cursor may be a blinking underscore or a reverse color block.) The other lines are marked with a ~ (tilde), the symbol for a non-existent line.

If, before entering **vi**, you have forgotten to set your terminal configuration or have set it to the wrong type of terminal, you will see an error message instead:

```
$ vi stuff<CR>
terminal_name: unknown terminal type

[Using open mode]
"stuff" [New file]
```

You cannot set the terminal configuration while you are in the editor; you must be in the shell. Leave the editor by typing:

:q<CR>

Then set the correct terminal configuration.

Creating Text: Append Mode

If you have successfully entered **vi**, you are in command mode and **vi** is waiting for your commands. How do you create text?

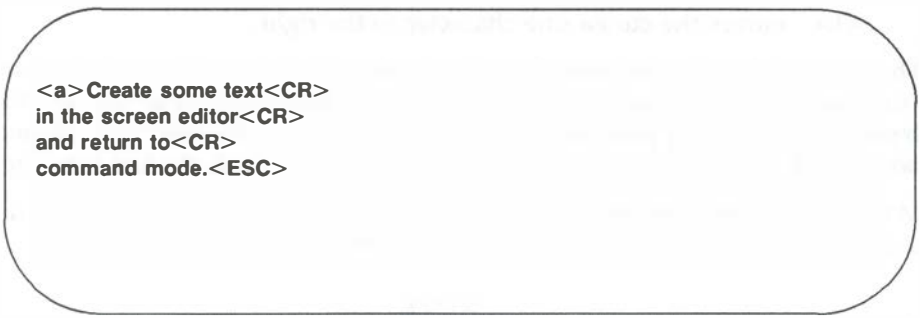
1. Press the **<A>** key (**<a>**) to enter the append mode of **vi**. (Do not press the **RETURN** key.) You can now add text to the file. (An A is not printed on the screen.)
2. Type in some text.

   As you approach the right margin, a bell sounds to remind you to press the **RETURN** key. (Terminals that do not have a bell may warn you in another way, e.g., by flashing the screen.)
3. To begin a new line, press the **RETURN** key.

If you have specified the **wrapmargin** option in your **.exrc** file, you get a new line whenever you get an automatic **RETURN**. (See *Setting the Automatic RETURN.*)

Leaving Append Mode

When you finish creating text, press the **ESCAPE** key to leave append mode and return to command mode. You can then edit any text you have created or write the text in the buffer to a file.



If you press the **ESCAPE** key and a bell sounds, you are already in command mode. The text in the file is not affected by this, even if you press the **ESCAPE** key several times.

Editing Text: Command Mode

To edit an existing file you must be able to add, change, and delete text. However, before you can perform those tasks, you must be able to move to the part of the file you want to edit. **vi** offers an array of commands for moving from page to page, between lines, and between specified points inside a line. These commands, along with commands for deleting and adding text, are introduced in this section.

Moving the Cursor

To edit your text, you need to move the cursor to the point on the screen where you will begin the correction. This is done with four keys that are grouped together on the keyboard:

<h> moves the cursor one character to the left.

<j> moves the cursor down one line.

<k> moves the cursor up one line.

<l> moves the cursor one character to the right.

The **<j>** and **<k>** commands maintain the column position of the cursor. For example, if the cursor is on the seventh character from the left, when you type **<j>** or **<k>**, it goes to the seventh character on the new line. If there is no seventh character on the new line, the cursor moves to the last character.

Many people who use **vi** find it helpful to mark these four keys with arrows showing the direction in which each key moves the cursor.

NOTE

Some terminals have special cursor control keys that are marked with arrows. Use them in the same way you use the **<h>**, **<j>**, **<k>**, and **<l>** commands.

Watch the cursor on the screen while you press the keys **<h>**, **<j>**, **<k>**, and **<l>**. Instead of pressing a motion command key a number of times to move the cursor a corresponding number of spaces or lines, you can precede the command with the desired number. For example, to move two spaces to the right, you can press **<l>** twice or enter **<2l>**. To move up four lines, press **<k>** four times or enter **<4k>**. If you cannot go any farther in the direction you have requested, **vi** will sound a bell.

Now experiment with the **j** and **k** motion commands. First, move the cursor up seven lines:

<7k>

The cursor will move up seven lines above the current line. If there are less than seven lines above the current line, a bell sounds and the cursor remains on the current line.

Now move the cursor down 35 lines:

<35j>

vi will clear and redraw the screen. The cursor will be on the 35th line below the current line, appearing in the middle of the new window. If there are less than 35 lines below the current line, the bell sounds and the cursor remains on the current line. Watch what happens when you type the next command:

<35k>

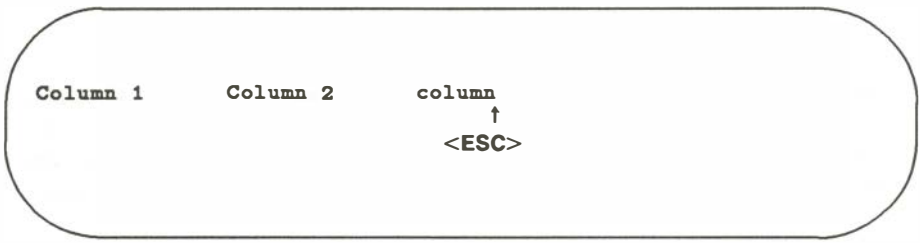
Like most **vi** commands, the **<h>**, **<j>**, **<k>**, and **<l>** motion commands are silent; they do not appear on the screen as you enter them. The only time you should see characters on the screen is when you are in append mode and are adding text to your file. If the motion command letters appear on the screen, you are still in append mode. Press the **ESCAPE** key to return to command mode and try the commands again.

In addition to the motion command keys **<h>** and **<l>**, the **SPACE BAR** and the **BACKSPACE** key can be used to move the cursor right or left to a character on the current line:

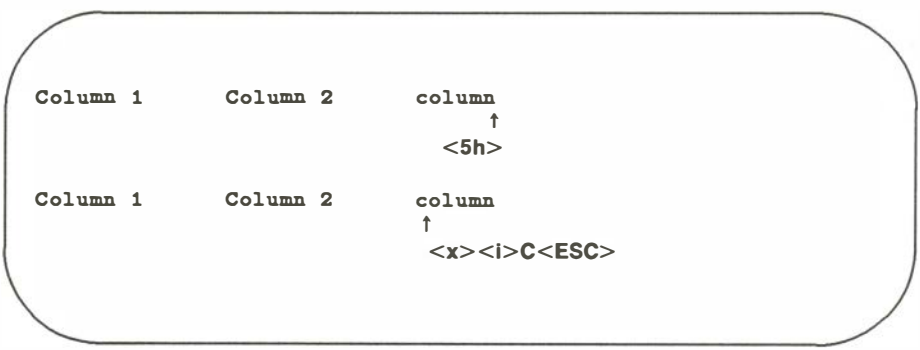
<SPACE BAR>	move the cursor one character to the right
<*n*SPACE BAR>	move the cursor *n* characters to the right
<BACKSPACE>	move the cursor one character to the left
<*n*BACKSPACE>	move the cursor *n* characters to the left

Try typing in a number before the command key. Notice that the cursor moves the specified number of characters to the left or right. In the following example, the cursor movement is shown by the arrows.

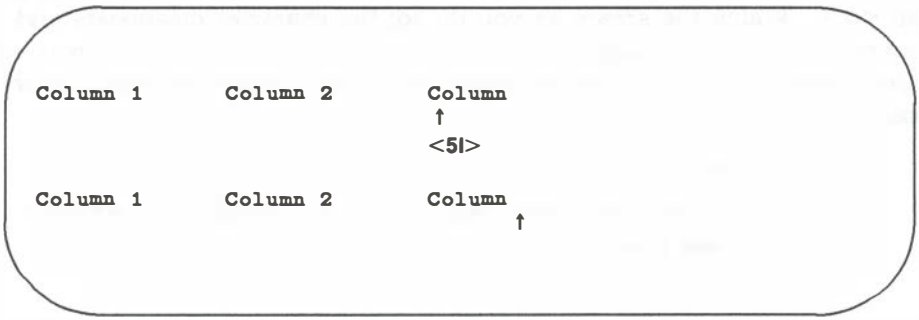
To move the cursor quickly to the right or left, prefix a number to the command. For example, suppose you want to create four columns in your screen. After you finish typing the headings for the first three columns, you notice a typing mistake.



You want to correct your mistake before continuing. Exit insert mode and return to command mode by pressing the **ESCAPE** key; the cursor moves to the **n**. Then use the <**h**> command to move back five spaces.



Erase the **c** by typing <**x**>. Then change to insert mode (<**i**>), enter a **C**, followed by the **ESCAPE** key. Use the <**l**> motion command to return to your earlier position.



By now you may have discovered that you can move the cursor back and forth on a line by using the space bar and the **BACKSPACE** key.

<SPACE BAR>	move the cursor one character to the right
<*n*SPACE BAR>	move the cursor *n* characters to the right
<BACKSPACE>	move the cursor one character to the left
<*n*BACKSPACE>	move the cursor *n* characters to the left

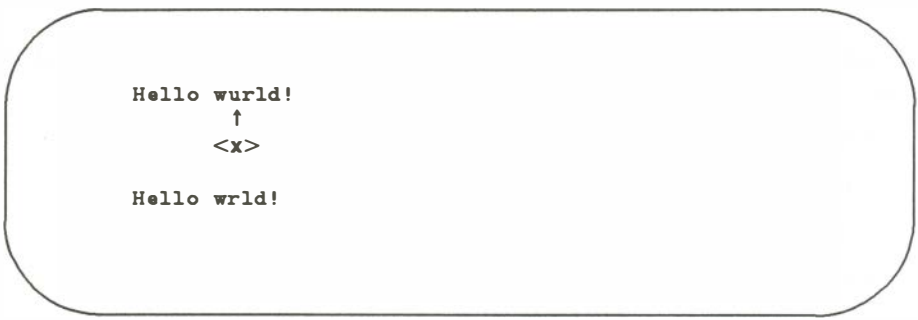
Again, you can specify a multiple space movement by typing a number before pressing the **SPACE BAR** or **BACKSPACE** key. The cursor moves the number of characters you request to the left or right.

Deleting Text

If you want to delete a character, move the cursor to that character and press the <x>. Watch the screen as you do so; the character disappears and the line readjusts to the change. To erase three characters in a row, press <x> three times. In the following example, the arrows under the letters show the positions of the cursor:

<x> delete one character

<*n*x> delete *n* characters, where *n* is the number of characters you want to delete



Now try preceding <x> with the number of characters you want to delete. For example, delete the second occurrence of the word **deep** from the text shown in the following screen. Put the cursor on the first letter of the string you want to delete, and delete five characters (for the four letters of **deep** plus an extra space).

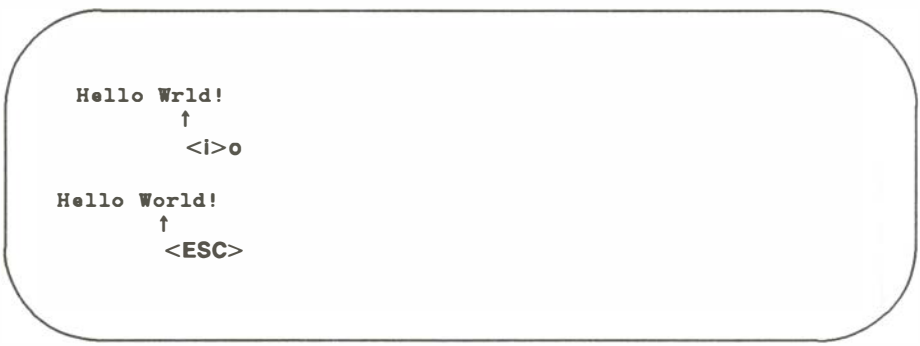
```
Tomorrow the Loch Ness monster
shall slither forth from
the deep dark deep depths of the lake.
                ↑
              <5x>
```

```
Tomorrow the Loch Ness monster
shall slither forth from
the deep dark depths of the lake.
                ↑
              <5x>
```

Notice that **vi** adjusts the text so that no gap appears in place of the deleted string. If, as in this case, the string you want to delete happens to be a word, you can also use the **vi** command for deleting a word. This command is described later in the section *Word Positioning*.

Adding Text

There are two basic commands for adding text: the insert (<**i**>) and append (<**a**>) commands. To add text with the insert command at a point in your file that is visible on the screen, move the cursor to that point by using <**h**>, <**j**>, <**k**>, and <**l**>. Then press <**i**> and start entering text. As you type, the new text appears on the screen to the left of the character on which you put the cursor. That character and all characters to the right of the cursor move right to make room for your new text. The **vi** editor continues to accept the characters you type until you press the **ESCAPE** key. If necessary, the original characters wrap around onto the next line.



You can use the append command in the same way. The only difference is that the new text appears to the right of the character on which you put the cursor.

Later in this tutorial you will learn how to move around on the screen or scroll through a file to add or delete characters, words, or lines.

Quitting vi

When you have finished your text, you will want to write the buffer contents to a file and return to the shell. To do this, hold down the SHIFT key and press <Z> twice (<ZZ>). The editor remembers the file name you specified with the **vi** command at the beginning of the editing session, and moves the buffer text to the file of that name. A notice at the bottom of the screen gives the file name and the number of lines and characters in the file. Then the shell gives you a prompt.

```
<a>This is a test file.<CR>
I am adding text to<CR>
a temporary buffer and<CR>
now it is perfect.<CR>
I want to write this file,<CR>
and return to the shell.<ESC><ZZ>
~
~
~
~
"stuff" [New file] 7 lines, 151 characters
$
```

You can also use the **:w** and **:q** commands of the line editor for writing and quitting a file. (Line editor commands begin with a colon and appear on the bottom line of the screen.) The **:w** command writes the buffer to a file. The **:q** command leaves the editor and returns you to the shell. You can type these commands separately or combine them into the single command **:wq**. It is easier to combine them.

```
<a>This is a test file.<CR>
I am adding text to<CR>
a temporary buffer and<CR>
now it is perfect.<CR>
I want to write this file,<CR>
and return to the shell.<ESC>
~
~
~
~
~
:wq<CR>
```

Table 6-1 summarizes the basic commands you need to enter and use **vi**.

Table 6-1. Summary of Commands for the **vi** Editor

Command	Function
TERM=*terminal_name* **export TERM**	set the terminal configuration
TermSetup	initialize the terminal as defined by *terminal_name*
vi *filename*	enter **vi** editor to edit the file called *filename*
<a>	add text after the cursor
<h>	move one character to the left
<j>	move down one line
<k>	move up one line
<l>	move one character to the right
<x>	delete a character
<CR>	carriage return
<ESC>	leave append mode, and return to **vi** command mode
:w	write to a file
:q	quit **vi**
:wq	write to a file and quit **vi**
<ZZ>	write to a file and quit **vi**

Exercise 1

Answers to the exercises are given at the end of this chapter. However, keep in mind that there is often more than one way to perform a task in **vi**. If your method works, it is correct.

As you give commands in the following exercises, watch the screen to see how it changes or how the cursor moves.

1-1. If you have not logged in yet, do so now. Then set your terminal configuration.

1-2. Enter **vi** and append the following five lines of text to a new file called **exer1**:

This is an exercise!
Up, down,
left, right,
build your terminal's
muscles bit by bit

1-3. Move the cursor to the first line of the file and the seventh character from the right. Notice that as you move up the file, the cursor moves in to the last letter of the file, but it does not move out to the last letter of the next line.

1-4. Delete the seventh and eighth characters from the right.

1-5. Move the cursor to the last character on the last line of the text.

1-6. Append the following new line of text:

and byte by byte

1-7. Write the buffer to a file and quit **vi**.

1-8. Re-enter **vi** and append two more lines of text to the file **exer1**. What does the notice at the bottom of the screen say once you have reentered **vi** to edit **exer1**?

Moving the Cursor Around the Screen

Until now you have been moving the cursor with the **<h>**, **<j>**, **<k>**, **<l>**, BACKSPACE key, and the SPACE BAR. There are several other commands that can help you move the cursor quickly around the screen. This section explains how to position the cursor in the following ways:

- by characters on a line
- by lines
- by text objects
  - — words
  - — sentences
  - — paragraphs
- in the window

There are also commands that position the cursor within parts of the **vi** editing buffer that are not visible on the screen. These commands will be discussed in the next section, *Scrolling the Text*.

To follow this section of the tutorial, you should enter **vi** with a file that contains at least 40 lines. If you do not have a file of that length, create one now. Remember, to execute the commands described here, you must be in command mode of **vi**. Press the ESCAPE key to make sure that you are in command mode instead of append mode.

Positioning the Cursor on a Character

There are three ways to position the cursor on a character in a line.

- by moving the cursor right or left to a character
- by specifying the character at either end of the line
- by searching for a character on a line

The first method was discussed earlier in this chapter under *Moving the Cursor Right or Left*. The following sections describe the other two methods.

Moving the Cursor to the Beginning or End of a Line

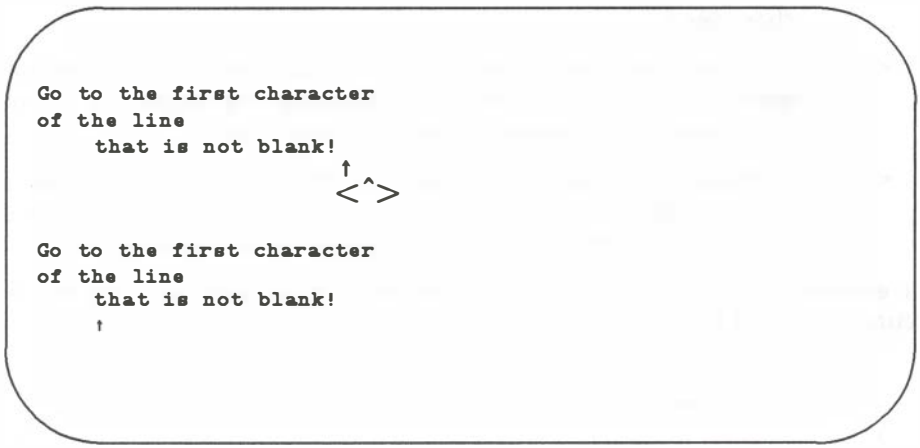
The second method of positioning the cursor on a line is by using one of three commands that put the cursor on the first or last character of a line.

<$>	puts the cursor on the last character of a line
<0> (zero)	puts the cursor on the first character of a line
<^> (circumflex)	puts the cursor on the first nonblank character of a line

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

```
Go to the end of the line!
          ↑
           <$>

Go to the end of the line!
                          ↑
```

```
Go to the beginning of the line!
                            ↑
                           <0>
Go to the beginning of the line!
↑
```



Searching for a Character on a Line

The third way to position the cursor on a line is to search for a specific character on the current line. If the character is not found on the current line, a bell sounds and the cursor does not move. (There is also a command that searches a file for patterns. This will be discussed in the next section.) There are six commands you can use to search within a line: **<f>**, **<F>**, **<t>**, **<T>**, <;>, and <,>. You must specify a character after all of them except the <;> and <,> commands.

<f*x***>** Move the cursor to the right to the specified character *x*.

<F*x***>** Move the cursor to the left to the specified character *x*.

<tx> Move the cursor right to the character just before the specified character *x*.

<Tx> Move the cursor left to the character just after the specified character *x*.

<;> Continue the search specified in the last command, in the same direction. The ; remembers the character and seeks out the next occurrence of that character on the current line.

<,> Continue the search specified in the last command, in the opposite direction. The , remembers the character and seeks out the previous occurrence of that character on the current line.

For example, in the following screen, **vi** searches to the right for the first occurrence of the letter A on the current line:

```
Go forward to the letter A on this line.
             ↑
          <fA>
Go forward to the letter A on this line.
                           ↑
```

Try the search commands on one of your files.

Moving the Cursor Up or Down a Line

Besides the <**j**> and <**k**> commands that you have already used, the <+>, <**-**>, and <**CR**> commands can be used to move the cursor to other lines.

The Minus Sign Motion Command

The <-> command moves the cursor up a line, positioning it at the first nonblank character on the line. To move more than one line at a time, specify the number of lines you want to move before the <-> command. For example, to move the cursor up 13 lines, type:

<13->

The cursor moves up 13 lines. If some of those lines are above the current window, the window scrolls up to reveal them. This is a rapid way to move quickly up a file.

Now try to move up 100 lines, type:

<100->

What happened to the window? If there are less then 100 lines above the current line a bell sounds, telling you that you have made a mistake; the cursor remains on the current line.

The Plus Sign Motion Command

The plus sign command (<+>) or the <CR> command moves the cursor down a line. Specify the number of lines you want to move before the <+> command. For example, to move the cursor down nine lines, type:

<9+>

The cursor will move down nine lines. If some of those lines are below the current screen, the window scrolls down to reveal them.

Now try to do the same thing by pressing the RETURN key. Were the results the same as when you pressed the + key?

Moving the Cursor to a Word

The **vi** editor considers a word to be a string of characters that may include letters, numbers, or underscores. There are six word positioning commands: <w>, <b>, <e>, <W>, <B>, and <E>. The lowercase commands (<w>, <b>, and <e>) treat any character other than a letter, digit, or underscore as a delimiter, signifying the beginning or end of a word. Punctuation before or after a blank is considered a word. The beginning or end of a line is also a delimiter.

The uppercase commands (<W>, <B>, and <E>) treat punctuation as part of the word; words are delimited by blanks and newlines only.

The following is a summary of the word positioning commands.

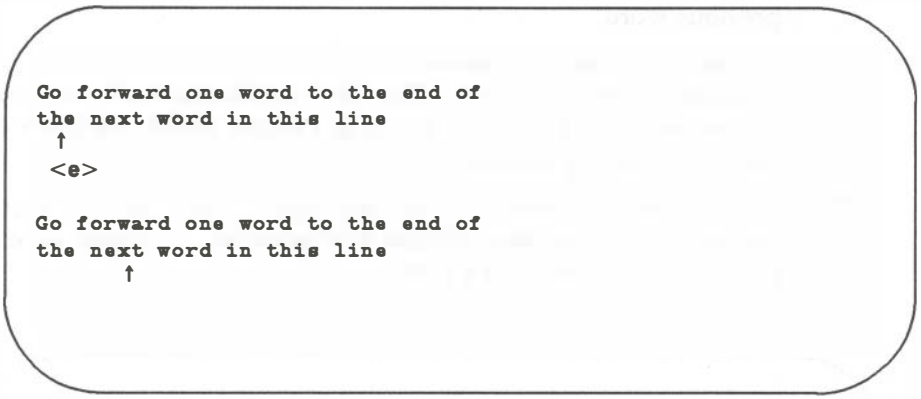
<w> Move the cursor forward to the first character in the next word. You may press <w> as many times as you want to reach the word you want, or you can prefix the necessary number to the <w>.

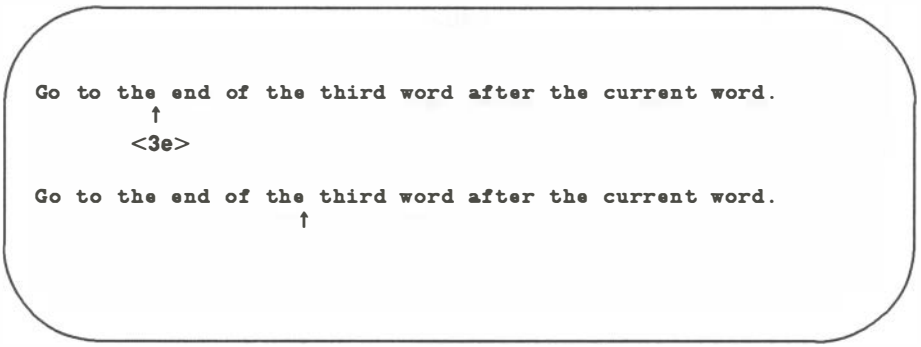
<*n*w> Move the cursor forward *n* number of words to the first character of that word. The end of the line does not stop the movement of the cursor; instead, the cursor wraps around and continues counting words from the beginning of the next line.

```
The <w> command
leaps word by word through the
file.  Move from THIS word forward
                  ↑
                 <6w>

six words to THIS word.
             ↑
```

<W> Ignores all punctuation and move the cursor forward to the word after the next blank.

<e> Moves the cursor forward in the line to the last character in the next word.


Go forward one word to the end of
the next word in this line
<e>
Go forward one word to the end of
the next word in this line


Go to the end of the third word after the current word.
<3e>
Go to the end of the third word after the current word.

<E> Ignores all punctuation except blanks, delimiting words only by blanks.

<b> Moves the cursor backward in the line to the first character of the previous word.

<*n*b> Moves the cursor backward *n* number of words to the first character of the *n*th word. The **<b>** command does not stop at the beginning of a line, but moves to the end of the line above and continues moving backward.

<B> Can be used just like the **<b>** command, except that it delimits the word only by blank spaces and newlines. It treats all other punctuation as letters of a word.


```
Leap backward word by word through
the file. Go back four words from here.
                                 ↑
                               <4b>


the file. Go back four words from here.
         ↑
```

Moving the Cursor by Sentences

The **vi** editor also recognizes sentences. In **vi**, a sentence ends in ! or . or ?. If these delimiters appear in the middle of a line, they must be followed by two blanks for **vi** to recognize them. You should get used to the **vi** convention of recognizing two blanks after a period as the end of a sentence, because it is often useful to be able to operate on a sentence as a unit.

You can move the cursor from sentence to sentence in the file with the <**(**> (open parenthesis) and <**)**> (close parenthesis) commands.

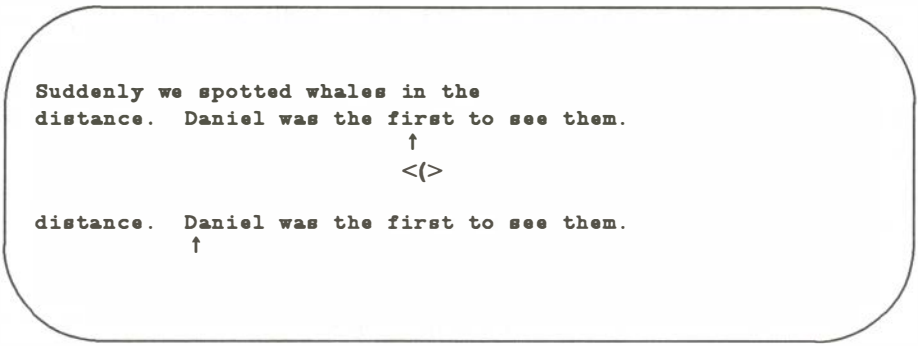
< **(** > Moves the cursor to the beginning of the current sentence.

< *n***(** > Moves the cursor to the beginning of the *n*th sentence above the current sentence.

< **)** > Moves the cursor to the beginning of the next sentence.

< *n***)** > Moves the cursor to the beginning of the *n*th sentence below the current sentence.

The example in the following screen shows how the open parenthesis moves the cursor around the screen.



Now repeat the command, preceding it with a number. For example, type:

<3(> (or)
<5)>

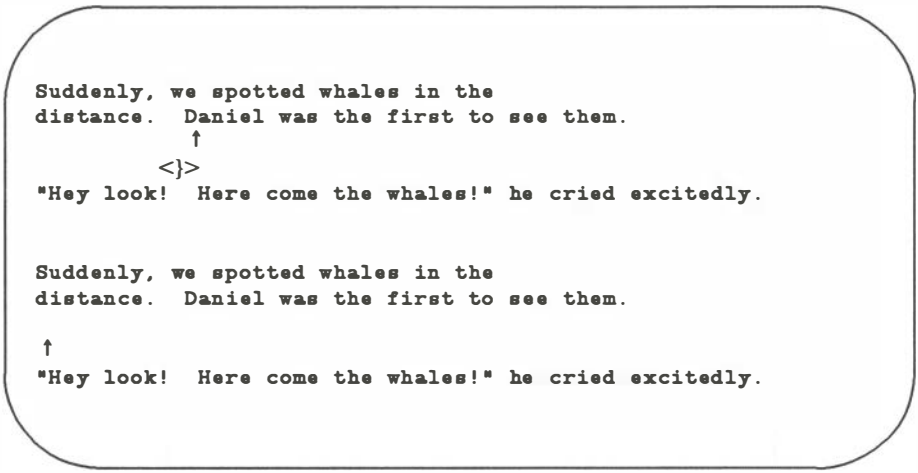
Did the cursor move the correct number of sentences?

Moving the Cursor by Paragraphs

Paragraphs are recognized by **vi** if they begin after a blank line. If you want to be able to move the cursor to the beginning of a paragraph (or later in this tutorial, to delete or change a whole paragraph), then make sure each paragraph ends in a blank line.

- **<{>** Moves the cursor to the beginning of the current paragraph, which is delimited by a blank line above it.
- **<n{>** Moves the cursor to the beginning of the *n*th paragraph above the current paragraph.
- **<}>** Moves the cursor to the beginning of the next paragraph.
- **<n}>** Moves the cursor to the *n*th paragraph below the current line.

The following screen show how the cursor can be moved to the beginning of another paragraph:



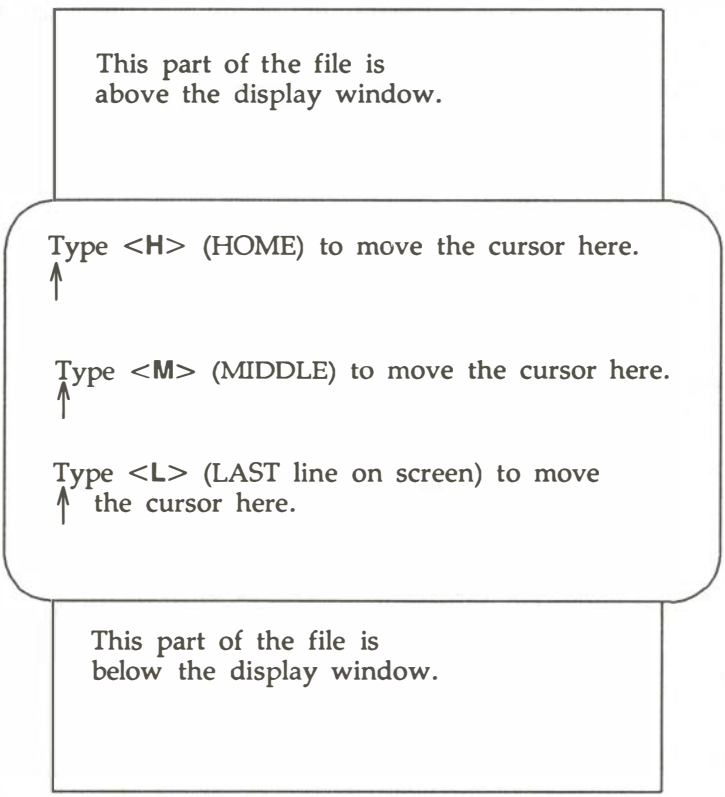
Moving the Cursor on the Screen

The **vi** editor also provides three commands that help you position yourself in the window. Try out each command. Be sure to type them in uppercase.

<**H**> Moves the cursor to the first line on the screen.

<**M**> Moves the cursor to the middle line on the screen.

<**L**> Moves the cursor to the last line on the screen.



Tables 6-2 through 6-5 summarize the **vi** commands for moving the cursor by positioning it on a character, line, word, sentence, paragraph, or position on the screen. (Additional **vi** commands for moving the cursor are summarized in Table 6-5, later in the chapter.)

Table 6-2. Summary of **vi** Motion Commands

Positioning on a Character	
<**h**>	Move the cursor one character to the left.
<**l**>	Move the cursor one character to the right.
<**BACKSPACE**>	Move the cursor one character to the left.
<**SPACE BAR**>	Move the cursor one character to the right.
<**f***x*>	Move the cursor to the right to the specified character *x*.
<**F***x*>	Move the cursor to the left to the specified character *x*.
<**t***x*>	Move the cursor to the right, to the character just before the specified character *x*.
<**T***x*>	Move the cursor to the left, to the character just after the specified character *x*.
<**;**>	Continue searching in same direction on the line for the last character requested with <**f**>, <**F**>, <**t**>, or <**T**>. The **;** remembers the character and finds the next occurrence of it on the current line.
<**,**>	Continue searching in opposite direction on the line for the last character requested with <**f**>, <**F**>, <**t**>, or <**T**>. The **,** remembers the character and finds the next occurrence of it on the current line.

Table 6-3. Summary of Commands for Positioning the Cursor on a Line

Positioning on a Line	
<k>	Move the cursor up to the same column in the previous line (if a character exists in that column).
<j>	Move the cursor down to the same column in the next line (if a character exists in that column).
<->	Move the cursor up to the beginning of the previous line.
<+>	Move the cursor down to the beginning of the next line.
<CR>	Move the cursor down to the beginning of the next line.

Table 6-4. Summary of Commands for Positioning the Cursor on a Word

Positioning on a Word	
<w>	Move the cursor forward to the first character in the next word.
<W>	Ignore all punctuation and move the cursor forward to the next word delimited only by blanks.
<b>	Move the cursor backward one word to the first character of that word.
<B>	Move the cursor to the left one word, which is delimited only by blanks.
<e>	Move the cursor to the end of the current word.
<E>	Delimit the words by blanks only. The cursor is placed on the last character before the next blank space, or end of the line.

NOTE: If the cursor is in the middle of a word, using these commands positions the cursor at the beginning or end of the current word.

Table 6-5. Summary of Commands for Positioning the Cursor on a Sentence, a Paragraph, or in a Window

Positioning on a Sentence	
<(>	Move the cursor to the beginning of the current sentence.
<)>	Move the cursor to the beginning of the next sentence.
Positioning on a Paragraph	
<{>	Move the cursor to the beginning of the current paragraph.
<}>	Move the cursor to the beginning of the next paragraph.
Positioning in the Window	
<H>	Move the cursor to the first line on the screen (the home position).
<M>	Move the cursor to the middle line on the screen.
<L>	Move the cursor to the last line on the screen.

Scrolling the Text

How do you move the cursor to text that is not shown in the current editing window? One option is to use the **<20j>** or **<20k>** command. However, if you are editing a large file, you need to move quickly and accurately to another place in the file. This section covers the commands that can help you move around within the file in the following ways:

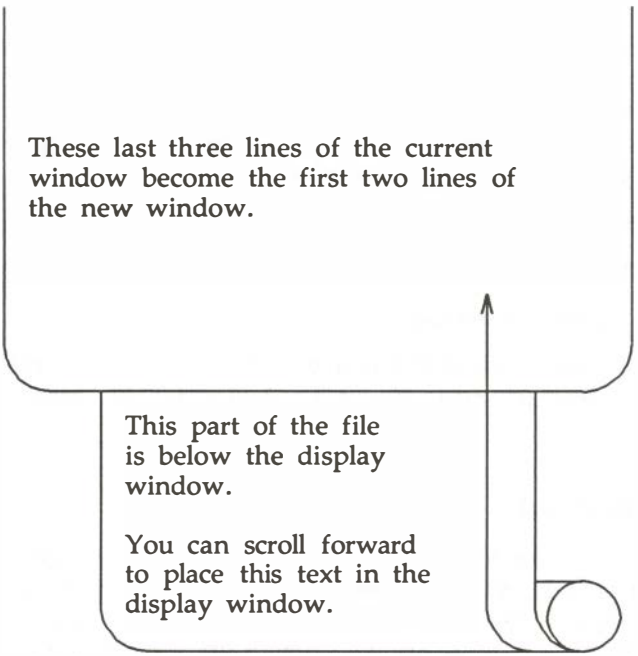
- by scrolling forward or backward in the file
- by going to a specified line in the file
- by searching for a pattern in the file

Four commands allow you to scroll the text of a file. The **<^f>** and **<^d>** commands scroll the screen forward. The **<^b>** and **<^u>** commands scroll the screen backward.

The Control-f Command

The **<^f>** command scrolls the text forward one full window of text below the current window. To do this, **vi** clears the screen and redraws the window. The three lines that were at the bottom of the current window are placed at the top of the new window. If there are not enough lines left in the file to fill the window, the screen displays a ~ (tilde) to show that there are empty lines.

vi clears and redraws the screen as follows:

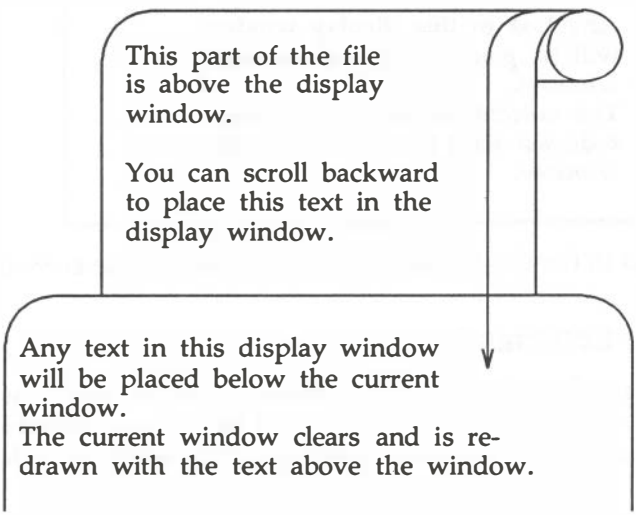


The Control-d Command

The <^**d**> command scrolls down a half screen to reveal text below the window. When you type <^**d**>, the text appears to be rolled up at the top and unrolled at the bottom. This allows the lines below the screen to appear on the screen, while the lines at the top of the screen disappear. If there are not enough lines in the file, a bell sounds (or the screen flashes).

The Control-b Command

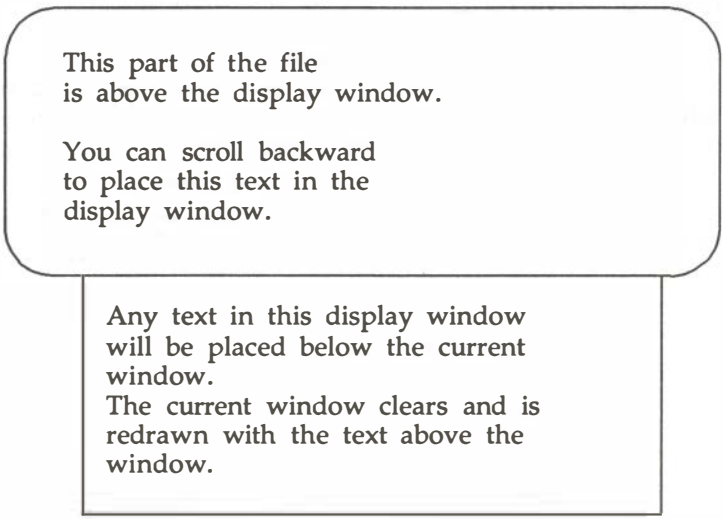
The <^**b**> command scrolls the screen back a full window to reveal the text above the current window. To do this, **vi** clears the screen and redraws the window with the text that is above the current screen. Unlike the <^**f**> command, <^**b**> does not leave any reference lines from the previous window. If there are not enough lines above the current window to fill a full new window, a bell sounds (or the screen flashes) and the current window remains on the screen.



Now try scrolling backward, type:

<^**b**>

vi clears the screen and draws a new screen.



Any text that was in the display window is placed below the current window.

The Control-u Command

The <^**u**> command scrolls up a half screen of text to reveal the lines just above the window. The lines at the bottom of the window are erased. Now scroll down in the text, moving the portion below the screen into the window, type:

<^**u**>

When the cursor reaches the top of the file, a bell sounds (or the screen flashes) to notify you that the file cannot scroll further.

Going to a Specific Line

The <**G**> command positions the cursor on a specified line in the window. If that line is not currently on the screen, <**G**> clears the screen and redraws the window around it. If you do not specify a line, <**G**> goes to the last line of the file.

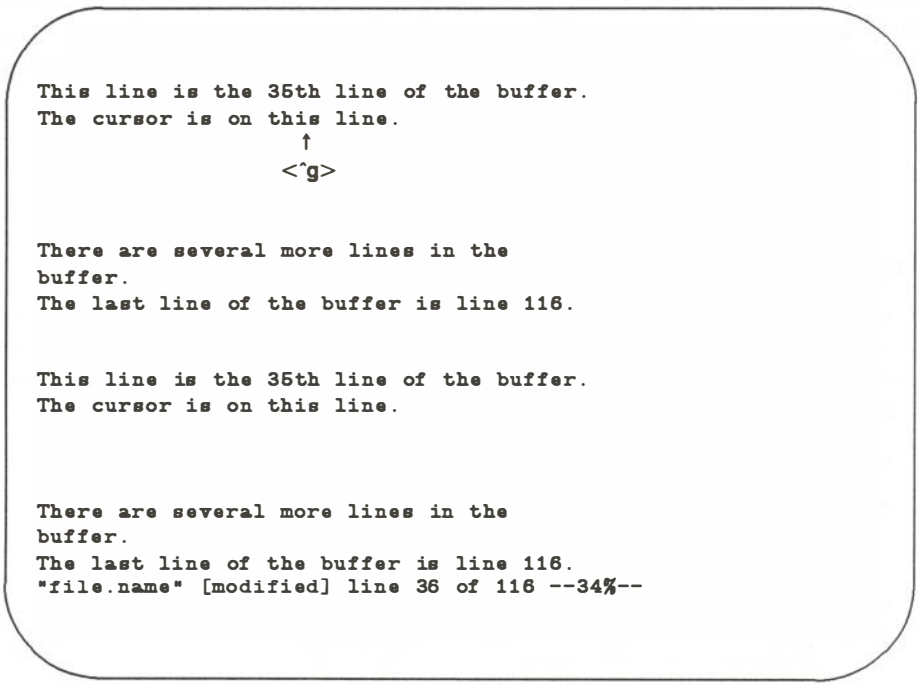
<**G**> go to the last line of the file

<*n***G**> go to the *n*th line of the file

Obtaining the Current Line Number

Each line of the file has a line number corresponding to its position in the buffer. To get the number of a particular line, position the cursor on it and type <^**g**>. The <^**g**> command gives you a status notice at the bottom of the screen which tells you:

- the name of the file
- if the file has been modified
- the line number on which the cursor rests
- the total number of lines in the buffer
- the percentage of the total lines in the buffer represented by the current line


This line is the 35th line of the buffer.
The cursor is on this line.
<^g>
There are several more lines in the
buffer.
The last line of the buffer is line 116.
This line is the 35th line of the buffer.
The cursor is on this line.
There are several more lines in the
buffer.
The last line of the buffer is line 116.
"file.name" [modified] line 36 of 116 --34%--

Searching for a Pattern

The fastest way to reach a specific place in your text is by using one of the search commands: **/**, **?**, **<n>**, or **<N>**. These commands allow you to search forward or backward in the buffer for the next occurrence of a specified character pattern. The **/** and **?** commands are not silent; they appear as you type them, along with the search pattern, on the bottom of the screen. The **<n>** and **<N>** commands, which allow you to repeat the requests you made for a search with a **/** or **?** command, are silent.

The **/**, followed by a *pattern* (*/pattern*), searches forward in the buffer for the next occurrence of the characters in *pattern*, and puts the cursor on the first of those characters. For example, the following command line finds the next occurrence in the buffer of the words **Hello world** and puts the cursor under the **H**.

/Hello world<CR>

The **?**, followed by a *pattern* (*?pattern*), searches backward in the buffer for the first occurrence of the characters in *pattern*, and puts the cursor on the first of those characters. For example, the following command line finds the last occurrence in the buffer (before your current position) of the words **data set design** and puts the cursor under the **d** in **data**.

?data set design<CR>

These search commands do not wrap around the end of a line while searching for two words. For example, you are searching for the words `Hello world`, if `Hello` is at the end of one line and `world` is at the beginning of the next, the search command does not find that occurrence of `Hello World`.

However, they do wrap around the end or the beginning of the buffer to continue a search. For example, if you are near the end of the buffer, and the pattern for which you are searching (with the */pattern* command) is at the top of the buffer, the command finds the pattern.

The **<n>** and **<N>** commands allow you to continue searches you have requested with */pattern* or **?***pattern* without retyping them.

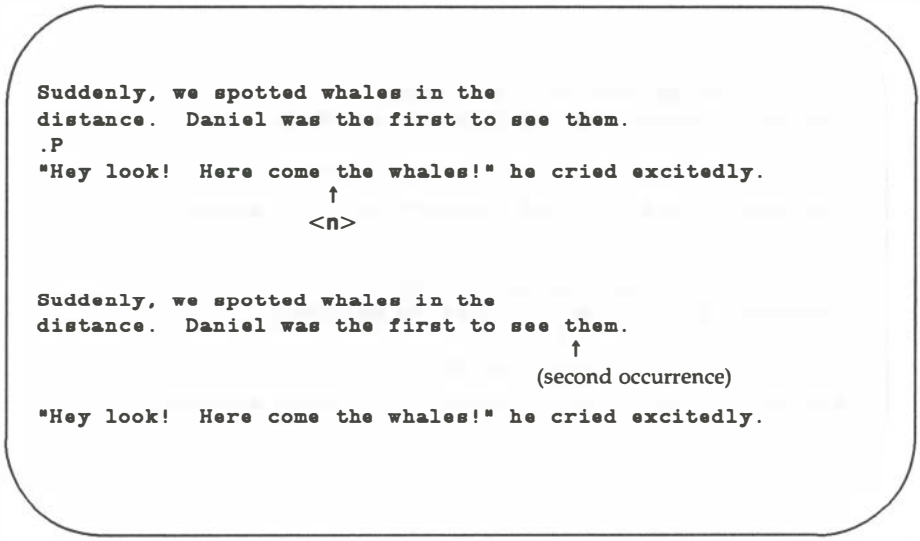
<n> Repeat the last search command.

<N> Repeat the last search command in the opposite direction.

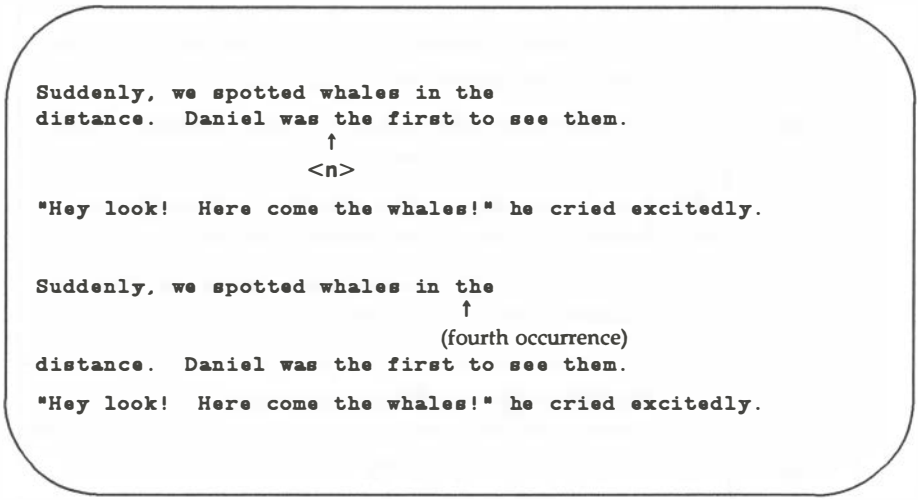
For example, you want to search backward in the file for the three-letter pattern **the**, initiate the search with **?the** and continue it with **<n>**.

The following screens offer a step-by-step illustration of how the **<n>** searches backward through the file and finds four occurrences of the character string **the**:

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

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



?the


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

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

                  (first occurrence)
```


Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
.P
"Hey look! Here come the whales!" he cried excitedly.
<n>
Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
(second occurrence)
"Hey look! Here come the whales!" he cried excitedly.

```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.
                                          ↑
                                        <n>

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


Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.
                      ↑
                (third occurrence)

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



The **/** and **?** search commands do not allow you to specify particular occurrences of a *pattern* with numbers. You cannot, for example, request the third occurrence (after your current position) of a *pattern*.

Table 6-6 summarizes the **vi** commands for moving the cursor by scrolling the text, specifying a line number, and searching for a *pattern*.

Table 6-6. Summary of Additional **vi** Motion Commands

Scrolling	
<^f>	Scroll the screen forward a full window, revealing the window of text below the current window.
<^d>	Scroll the screen down a half window, revealing lines below the current window.
<^b>	Scroll the screen back a full window, revealing the window of text above the current window.
<^u>	Scroll the screen up a half window, revealing the lines of text above the current window.
Positioning on a Numbered Line	
<1G>	Go to the first line of the file.
<G>	Go to the last line of the file.
<^g>	Give the line number and file status.
Searching for a Pattern	
/*pattern*	Search forward in the buffer for the next occurrence of the *pattern*. Position the cursor on the first character of the *pattern*.
?*pattern*	Search backward in the buffer for the first occurrence of the *pattern*. Position the cursor under the first character of the *pattern*.
<n>	Repeat the last search command.
<N>	Repeat the search command in the opposite direction.

Exercise 2

2-1. Create a file called **exer2**. Type a number on each line, numbering the lines from 1 to 50. Your file should look similar to the following:

```
 1
 2
 3
 .
 .
 .
48
49
50
```

2-2. Try using each of the scroll commands, noticing how many lines scroll through the window. Try the following:

<^f>
<^b>
<^u>
<^d>

2-3. Go to the end of the file. Append the following line of text:

```
123456789 123456789
```

What number does the command **<7h>** place the cursor on? What number does the command **<3l>** place the cursor on?

2-4. Try the command **<$>** and the command **<0>** (number zero).

2-5. Go to the first character on the line that is not a blank. Move to the first character in the next word. Move back to the first character of the word to the left. Move to the end of the word.

2-6. Go to the first line of the file. Try the commands that place the cursor in the middle of the window, on the last line of the window, and on the first line of the window.

2-7. Search for the number 8. Find the next occurrence of the number 8. Find 48.

Creating Text

There are three basic commands for creating text:

<**a**> append text

<**i**> insert text

<**o**> open a new line on which text can be entered

After you finish creating text with any one of these commands, you can return to the command mode of **vi** by pressing the ESCAPE key.

Appending Text

<**a**> append text after the cursor

<**A**> append text at the end of the current line

You have already experimented with the <**a**> command in the *Creating a File* section. Make a new file named **junk2**. Append some text using the <**a**> command. To return to command mode of **vi**, press the ESCAPE key, then compare the <**a**> command to the <**A**> command.

Inserting Text

<**i**> insert text before the cursor

<**I**> insert text at the beginning of the current line before the first character that is not a blank

To return to the command mode of **vi**, press the **ESCAPE** key.

In the following examples, you can compare the append and insert commands. The arrows show the position of the cursor, where new text will be added.

```
Append three spaces AFTER the H of Here
                                    ↑
                                   <a>

Append three spaces AFTER the H of H   ere.
                                     ↑
                                   <ESC>
```

```
Insert three spaces BEFORE the H of Here.
                                    ↑
                                    <i>.

Insert three spaces BEFORE the H of   Here.
                                     ↑
                                   <ESC>
```

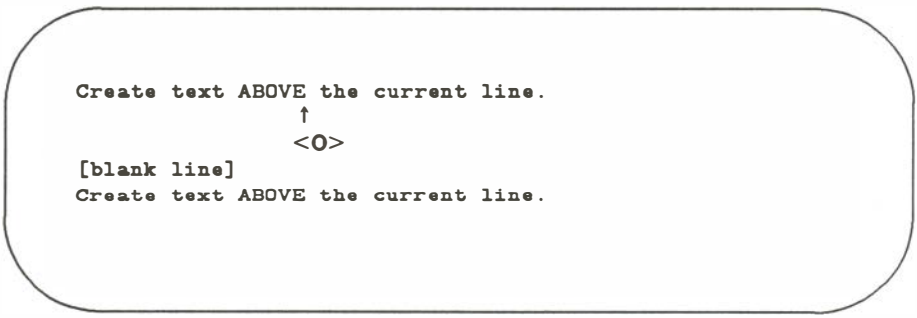
Notice that in both cases the user has left text input mode by pressing the **ESCAPE** key.

Opening a Line for Text

<o> Creates text from the beginning of a new line below the current line. You can issue this command from any point in the current line.

<O> Creates text from the beginning of a new line above the current line. This command can also be issued from any position in the current line.

The open command creates a directly above or below the current line, and puts you into text input mode. For example, in the following screens the **<O>** command opens a line above the current line, and the **<o>** command opens a line below the current line. In both cases, the cursor waits for you to enter text from the beginning of the new line.



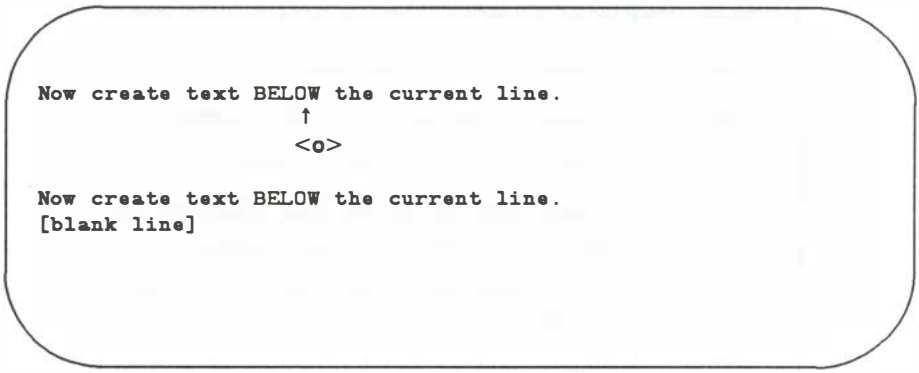


Table 6-7 summarizes the commands for creating and adding text with the **vi** editor.

Table 6-7. Summary of **vi** Commands for Creating Text

Command	Function
<a>	Create text after the cursor.
<A>	Create text at the end of the current line.
<i>	Create text in front of the cursor.
<I>	Create text before the first character on the current line that is not a blank.
<o>	Create text at the beginning of a new line below the current line.
<O>	Create text at the beginning of a new line above the current line.
<ESC>	Return **vi** to command mode from any of the above text input modes.

Exercise 3

3-1. Create a text file called **exer3**.

3-2. Insert the following four lines of text:

Append text
Insert text
a computer's
job is boring.

3-3. Add the following line of text above the last line:

financial statement and

3-4. Using a text insert command, add the following line of text above the third line:

Delete text

3-5. Add the following line of text below the current line:

byte of the budget

3-6. Using an append command, add the following line of text below the last line:

But, it is an exciting machine.

3-7. Move to the first line and add the word some before the word text.

Now practice using each of the six commands for creating text.

3-8. Leave **vi** and go on to the next section to find out how to delete any mistakes you made in creating text.

Deleting Text

You can delete text with various commands in command mode, and undo the entry of small amounts of text in text input mode. In addition, you can undo entirely the effects of your most recent command.

Undoing Entered Text in Text Input Mode

To delete a character at a time when you are in text input mode use the **BACKSPACE** key.

<BACKSPACE> Deletes the character under the cursor.

The **BACKSPACE** key backs up the cursor in text input mode and deletes each character that the cursor backs across. However, the deleted characters are not erased from the screen until you type over them or press the **ESCAPE** key to return to command mode.

In the following example, the arrows represent the cursor:

```
Mary had a litttl
                 ↑
               <BACKSPACE> <BACKSPACE>

Mary had a litttl
               ↑
             <ESC>


Mary had a litt
             ↑
```

Notice that characters are not erased from the screen until you press the **ESCAPE** key.

There are two other keys that delete text in text input mode. Although you may not use them often, you should be aware that they are available. To remove the special meanings of these keys so that they can be typed as text, see the section on special commands.

<^W> undo the entry of the current word

<CKILL> delete all text entered on current line since text input mode was entered

When you type <^w>, the cursor backs up over the word last typed and stops on its first character. It does not erase the word until you press the ESCAPE key or enter new characters over the old ones. The <CKILL> behaves in a similar way except that it removes all text you have typed on the current line since you last entered input mode.

Undoing the Last Command

Before you experiment with the delete commands, you should try the <u> command. This command undoes the last command you issued:

<u> undoes the last command.

<U> restores the current line to its state before you changed it.

If you delete lines by mistake, type <u>; your lines reappear on the screen. If you type the wrong command, type <u> and it is nullified. The <U> command nullifies all changes made to the current line as long as the cursor has not been moved from it.

If you type <u> twice in a row, the second command undos the first; your undo is undone! For example, you delete a line by mistake and restore it by typing <u>. Typing <u> a second time deletes the line again. Knowing this command can save you a lot of trouble.

The Delete Commands

You know that you can precede a command by a number. Many of the commands in **vi**, such as the delete and change commands, also allow you to enter a cursor movement command after another command. The cursor movement command can specify a text object such as a word, line, sentence, or paragraph. The general format of a **vi** command is:

[*number*][*command*]*text_object*

The brackets around components of the command format show that those components are optional.

All delete commands issued in command mode immediately remove unwanted text from the screen and redraw the affected part of the screen.

The delete command follows the general format of a **vi** command:

[*number*]**d***text_object*

NOTE

On some terminals the deleted lines are replaced by @ signs until the screen is redrawn by scrolling or command. The @ sign is not part of your file.

Deleting Characters

You can delete a character by moving the cursor to that character and pressing <**x**>. The character under the cursor is erased and the line readjusts to the change. You can delete several characters at once by specifying a number before the command.

Deleting Words

You can delete a word or part of a word with the <**dw**> command. Move the cursor to the first character to be deleted and type <**dw**>. The character under the cursor and all subsequent characters in that word are erased.

```
the deep dark depths of the lake.
    ↑
    <2dw>


the depths of the lake.
   ↑
```

The <**dw**> command deletes one word or punctuation mark and the space(s) that follow it. You can delete several words or marks at once by specifying a number before the command. For example, to delete three words and two commas, type <**5dw**>:

```
the deep, deep, dark depths of the lake
    ↑
    <5dw>


the depths of the lake
    ↑
```

Deleting Paragraphs

To delete paragraphs, use the following commands.

<**d**{> or <**d**}>

Observe what happens to your file. Remember, you can restore the deleted text with <**u**>.

Deleting Lines

To delete a line, type <**dd**>. To delete multiple lines, specify a number before the command. For example, typing the following erases 10 lines:

<**10dd**>

If you delete more than a few lines, **vi** displays the following notice on the bottom of the screen:

```
10 lines deleted
```

If there are less than 10 lines below the current line in the file, a bell sounds (or the screen flashes) and no lines are deleted.

Deleting Text After the Cursor

To delete all text on a line after the cursor, put the cursor on the first character to be deleted and type:

<**D**> or <**d$**>.

Neither of these commands allows you to specify a number of lines; they can be used only on the current line.

Table 6-8 summarizes the **vi** commands for deleting text.

Table 6-8. Summary of Delete Commands

Command	Function
For Insert Mode:	
<BACKSPACE>	Delete the current character.
<^h>	Delete the current character.
<^W>	Delete the current word.
<@>	Delete the current line of new text or delete all new text on the current line.
For Command Mode:	
<u>	Undo the last command.
<U>	Restore current line to its previous state.
<x>	Delete the current character.
<*n*d*x*>	Delete *n* number of text objects of type *x*.
<dw>	Delete the word at the cursor through the next space or to the next punctuation mark.
<dW>	Delete the word and punctuation at the cursor through the next space.
<dd>	Delete the current line.
<D>	Delete the portion of the line to the right of the cursor.
<d)>	Delete the current sentence.
<d}>	Delete the current paragraph.

Exercise 4

4-1. Create a file called **exer4** and put the following four lines of text in it:

When in the course of human events
there are many repetitive, boring
chores, then one ought to get a
robot to perform those chores.

4-2. Move the cursor to line two and append to the end of that line:

tedious and unsavory.

Delete the word unsavory while you are in append mode.

Delete the word boring while you are in command mode.

What is another way you could have deleted the word boring?

4-3. Insert at the beginning of line four:

congenial and computerized.

Delete the line.

How can you delete the contents of the line without removing the line itself?

Delete all the lines with one command.

4-4. Leave the screen editor and remove the empty file from your directory.

Modifying Text

The delete commands and text input commands provide one way for you to modify text. Another way you can change text is by using a command that lets you delete and create text simultaneously. There are three basic change commands: <r>, <s>, and <c>.

Replacing Text

<r> Replaces the current character (the character shown by the cursor). This command does not initiate text input mode and so does not need to be followed by pressing the ESCAPE key.

<*n*r> Replaces *n* characters with the same letter. This command automatically terminates after the *nth* character is replaced. It does not need to be followed by pressing the ESCAPE key.

<R> Replaces only those characters typed over until the ESCAPE command is given. If the end of the line is reached, this command will append the input as new text.

The <r> command replaces the current character with the next character that is typed in. For example, suppose you want to change the word acts to ants in the following sentence:

```
The circus has many acts.
```

Place the cursor under the `c` of `acts` and type

```
<r>n
```

The sentence becomes:

```
The circus has many ants.
```

To change `many` to `7777`, place the cursor under the `m` of `many` and type:

```
<4r7>
```

The <r> command changes the four letters of `many` to four occurrences of the number seven:

```
The circus has 7777 ants.
```

Substituting Text

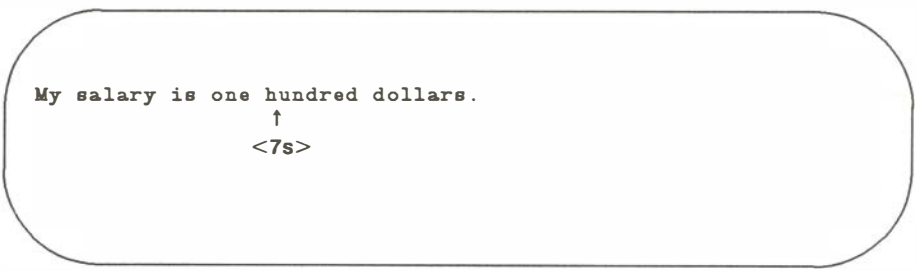
The substitute command replaces characters, but then allows you to continue to insert text from that point until you press the **ESCAPE** key.

- **<s>** Deletes the character shown by the cursor and append text. End the text input mode by pressing the **ESCAPE** key.
- **<*n*s>** Deletes *n* characters and append text. End the text input mode by pressing the **ESCAPE** key.
- **<S>** Replaces all the characters in the line.

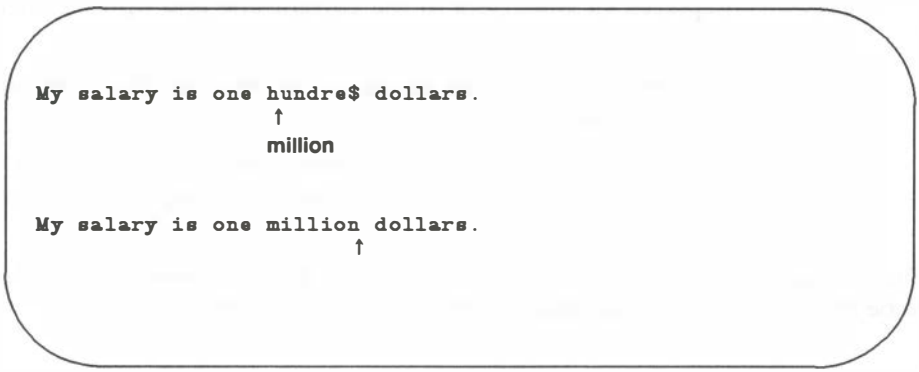
When you enter the **<s>** command, the last character in the string of characters to be replaced is overwritten by a **$** sign. The characters are not erased from the screen until you type over them, or leave text input mode by pressing the **ESCAPE** key.

Notice that you cannot use an argument with either **<r>** or **<s>**. Did you try?

Suppose you want to substitute the word million for the word hundred in the sentence `My salary is one hundred dollars.` Put the cursor under the `h` of `hundred` and type **<7s>**. Notice where the **$** sign appears.



Then type **million**:



Changing Text

The substitute command replaces characters. The change command replaces text objects, and then continues to append text from that point until you press the **ESCAPE** key. To end the change command, press the **ESCAPE** key.

The change command can take an argument. You can replace a character, word, or an entire line with new text.

<*n***c***x*> Replaces *n* number of text objects of type *x*, such as sentences (shown by <**)**>) and paragraphs (shown by <}>).

<**cw**> Replaces a word or the remaining characters in a word with new text. The **vi** editor prints a **$** sign to show the last character to be changed.

<*n*cw> Replaces *n* words.

<cc> Replaces all the characters in the line.

<*n*cc> Replaces all characters in the current line and up to *n* lines of text.

<C> Replaces the remaining characters in the line, from the cursor to the end of the line.

<*n*C> Replaces the remaining characters from the cursor in the current line and replace all the lines following the current line up to *n* lines.

The change commands, <cw> and <C>, use a $ sign to mark the last letter to be replaced. Notice how this works in the following example:

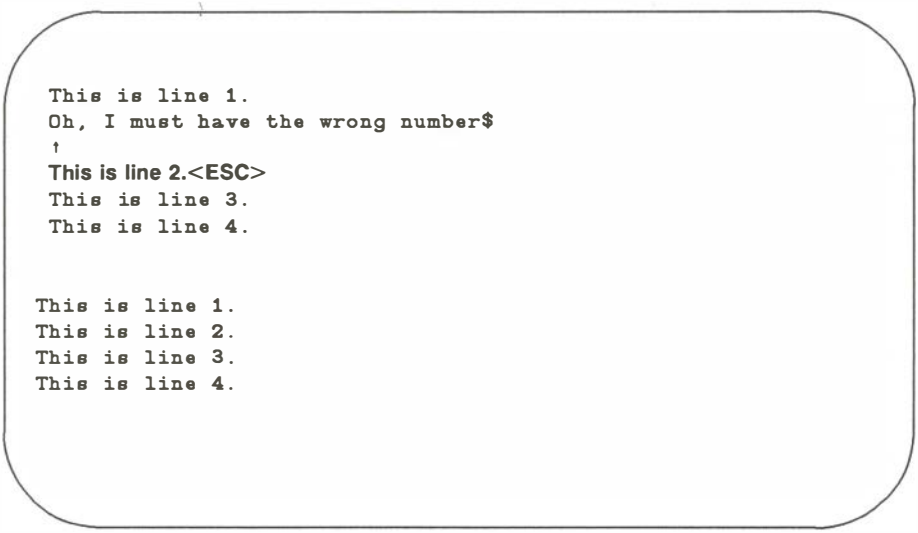
```
They are now due to arrive on Tuesday.
                              ↑
                              <cw>
```

```
They are now due to arrive on Tuesda$.
                              ↑
                              Wednesday<ESC>

They are now due to arrive on Wednesday.
                                       ↑
```

Notice that the new word (**Wednesday**) has more letters than the word it replaced (**Tuesday**). Once you have executed the change command, you are in text input mode and can enter as much text as you want. The buffer accepts text until you press the **ESCAPE** key.

The <**C**> command, when used to change the remaining text on a line, works in the same way. When you enter the command, it uses a **$** sign to mark the end of the text that will be deleted, puts you in text input mode, and waits for you to type new text over the old. The following screens offer an example of the **C** command:

```
This is line 1.
Oh, I must have the wrong number.
 ↑
<C>
This is line 3.
This is line 4.
```



Now try combining arguments. For example, type:

<c{>

Because you know the undo command, do not hesitate to experiment with different arguments or to precede the command with a number. You must press the ESCAPE key before using the <**u**> command, since <**c**> places you in text input mode. Compare <**S**> and <**cc**>. The two commands should produce the same results.

Table 6-9 summarizes the **vi** commands for changing text.

Table 6-9. Summary of **vi** Commands for Changing Text

Command	Function
<**r**>	Replace the current character.
<**R**>	Replace only those characters typed over with new characters until the ESCAPE key is pressed.
<**s**>	Delete the character the cursor is on and append text. End the append mode by pressing the ESCAPE key.
<**S**>	Replace all characters in the line.
<**cc**>	Replace all characters in the line.
<*n***c***x*>	Replace *n* number of text objects of type *x*, such as sentences (shown by <**)**>) and paragraphs (shown by <}>).
<**cw**>	Replace a word or the remaining characters in a word with new text.
<**C**>	Replace the remaining characters in the line, from the cursor to the end of the line.

Cutting and Pasting Text Electronically

vi provides a set of commands that cut and paste text in a file. Another set of commands copies a portion of text and places it in another section of a file.

Moving Text

You can move text from one place to another in the **vi** buffer by deleting the lines and then placing them at the required point. The last text that was deleted is stored in a temporary buffer. If you move the cursor to that part of the file where you want the deleted lines to be placed and press the <**p**> key, the deleted lines are added below the current line:

<**p**> Place the contents of the temporary buffer after the cursor.

A partial sentence that was deleted by the <**D**> command can be placed in the middle of another line. Position the cursor in the space between two words, then press <**p**>. The partial line is placed after the cursor.

Characters deleted by <*n***x**> also go into a temporary buffer. Any text object that was just deleted can be placed somewhere else in the text with <**p**>.

The <**p**> command should be used right after a delete command since the temporary buffer only stores the results of one command at a time. The <**p**> command is also used to copy text placed in the temporary buffer by the yank command. The yank command (<**y**>) is discussed in *Copying Text*.

Fixing Transposed Letters

A quick way to fix transposed letters is to combine the <**x**> and the <**p**> commands as <**xp**>. <**x**> deletes the letter. <**p**> places it after next character.

Notice the error in the next line:

```
A line of tetx
```

This error can be changed quickly by placing the cursor under the **t** in **tx** and then pressing the <**x**> and <**p**> keys, in that order. The result is:

```
A line of text
```

Try this. Make a typing error in your file and use the <**xp**> command to correct it. Why does this command work?

Copying Text

You can yank (copy) one or more lines of text into a temporary buffer, and then put a copy of that text anywhere in the file. To put the text in a new position type <**p**>; the text appears on the next line.

The yank command follows the general format of a **vi** command:

[*number*]**y**[*text_object*]

Yanking lines of text does not delete them from their original position in the file. If you want the same text to appear in more than one place, this provides a convenient way to avoid typing the same text several times. However, if you do not want the same text in multiple places, be sure to delete the original text after you have put the text into its new position.

Table 6-10 summarizes the ways you can use the yank command.

Table 6-10. Summary of the Yank Command

Command	Function
<*n***y***x*>	Yank *n* number of text objects of type *x*, (such as sentences **)** and paragraphs }).
<**yw**>	Yank a copy of a word.
<**yy**>	Yank a copy of the current line.
<*n***yy**>	Yank *n* lines.
<**y)**>	Yank all text up to the end of a sentence.
<**y}**>	Yank all text up to the end of the paragraph.

Notice that this command allows you to specify the number of text objects to be yanked.

Try the following command lines and see what happens on your screen. (Remember, you can always undo your last command.) Type:

<**5yw**>

Move the cursor to another spot, type:

<**p**>

Now try yanking a paragraph <**y}**> and placing it after the current paragraph. Then move to the end of the file <**G**> and place that same paragraph at the end of the file.

Copying or Moving Text Using Registers

Moving or copying several sections of text to a different part of the file is tedious work. **vi** provides a shortcut for this: named registers in which you can store text until you want to move it. To store text you can either yank or delete the text you wish to store.

Using registers is useful if a piece of text must appear in many places in the file. The extracted text stays in the specified register until you either end the editing session, or yank or delete another section of text to that register.

The general format of the command is:

[*number*]["*x*]*command*[*text_object*]

The *x* is the name of the register and can be any single letter. It must be preceded by a double quotation mark. For example, place the cursor at the beginning of a line, type:

<3"ayy>

Type in more text and then go to the end of the file, type:

<"ap>

Did the lines you saved in register **a** appear at the end of the file?

Table 6-11 summarizes the cut and paste commands.

Table 6-11. Summary of **vi** Commands for Cutting and Pasting Text

Command	Function
<**p**>	Place the contents of the temporary buffer containing the text obtained from the most recent delete or yank command into the text after the cursor.
<**yy**>	Yank a line of text and place it into a temporary buffer.
<*n***y***x*>	Yank a copy of *n* number of text objects of type *x* and place them in a temporary buffer.
<"x**y***n*>	Place a copy of a text object of type *n* in the register named by the letter x.
<"x**p**>	Place the contents of the register x after the cursor.

Exercise 5

5-1. Enter **vi** with the file called **exer2** that you created in Exercise 2. Go to line eight and change its contents to:

END OF FILE

5-2. Yank the first eight lines of the file and place them in register **z**. Put the contents of register **z** after the last line of the file.

5-3. Go to line eight and change its contents to:

eight is great

5-4. Go to the last line of the file. Substitute **EXERCISE** for `FILE`. Replace `OF` with **TO**.

Special Commands

Here are some special commands that you will find useful.

<.> repeat the last command

<J> join two lines together

<^l> clear the screen and redraw it

<~> change lowercase to uppercase and vice versa

Repeating the Last Command

The . period repeats the last command to create, delete, or change text in the file. It is often used with the search command.

For example, suppose you forget to capitalize the S in United States. However, you do not want to capitalize the s in chemical states. One way to correct this problem is by searching for the word states. The first time you find it in the expression United States, you can change the s to S. Then continue your search. When you find another occurrence, you can simply type a period; **vi** will remember your last command and repeat the substitution of s for S.

Experiment with this command. For example, if you try to add a period at the end of a sentence while in command mode, the last text change suddenly appears on the screen. Watch the screen to see how the text is affected.

Joining Two Lines

The <**J**> command joins lines. To enter this command, place the cursor on the current line, and press the <**SHIFT**> and **j** keys simultaneously. The current line is joined with the following line.

For example, suppose you have the following two lines of text:

```
Dear Mr.
Smith:
```

To join these two lines into one, place the cursor under any character in the first line and type:

<J>

You will immediately see the following on your screen:

```
Dear Mr. Smith:
```

Notice that **vi** automatically places a space between the last word on the first line and the first word on the second line.

Clearing and Redrawing the Window

If another operating system user sends you a message using the write command while you are editing with **vi**, the message appears in your current window, over part of the text you are editing. To restore your text after you read the message, you must be in command mode. (If you are in text input mode, press the ESCAPE key to return to command mode.) Then type <^l> (control-l). **vi** will erase the message and redraw the window exactly as it appeared before the message arrived.

Changing Lowercase to Uppercase and Vice Versa

A quick way to change any lowercase letter to uppercase, or vice versa, is by putting the cursor on the letter to be changed and typing a <~> (tilde). For example, to change the letter a to A, press ~. You can change several letters by typing ~ several times, but you cannot precede the command with a number to change several letters with one command.

Table 6-12 summarizes the special commands.

Table 6-12. Summary of Special Commands

Command	Function
<.>	Repeat the last command.
<J>	Join the line below the current line with the current line.
<^l>	Clear and redraw the current window.
<~>	Change lower case to upper case, or vice versa.

Using Line Editing Commands in vi

The **vi** editor has access to many of the commands provided by a line editor called **ex**. (For a complete list of **ex** commands see the **ex**(1) page in the *User's Reference Manual*.) This section discusses some of those most commonly used.

The **ex** commands are very similar to the **ed** commands discussed in Chapter 5. If you are familiar with **ed**, you may want to experiment on a test file to see how many **ed** commands also work in **vi**.

Line editor commands begin with a **:** (colon). After the colon is typed, the cursor will drop to the bottom of the screen and display the colon. The remainder of the command also appears at the bottom of the screen as you type it.

Temporarily Returning to the Shell

When you enter **vi**, the contents of the buffer fill your screen, making it impossible to issue any shell commands. However, you may want to do so. For example, you may want to get information from another file to incorporate into your current text. You could get that information by running one of the shell commands that display the text of a file on your screen, such as the **cat** or **pg** command. However, quitting and re-entering the editor is time consuming and tedious. **vi** offers two methods of escaping the editor

temporarily so that you can issue shell commands (and even edit other files) without having to write your buffer and quit: the **:!** command and the **:sh** command.

The **:!** command allows you to escape the editor and run a shell command on a single command line. From the command mode of **vi**, type **:!**. These characters will be printed at the bottom of your screen. Type a shell command immediately after the **!**. The shell will run your command, give you output, and print the message `[Hit return to continue]`. When you press the RETURN key, **vi** refreshes the screen and the cursor reappears exactly where you left it.

The **ex** command **:sh** allows you to do the same thing, but behaves differently on the screen. From the command mode of **vi** type **:sh** and press the RETURN key. A shell command prompt appears on the next line. Type your command(s) after the prompt as you would normally do while working in the shell. When you are ready to return to **vi**, type <^**d**> or **exit**; your screen is refreshed with your buffer contents and the cursor appears where you left it.

Even changing directories while you are temporarily in the shell will not prevent you from returning to the **vi** buffer where you were editing your file when you type **exit** or <^**d**>.

Writing Text to a New File: :w Command

The **:w** (for write) command allows you to create a file by copying lines of text from the file you are currently editing into a file that you specify. To create your new file you must specify a line or range of lines (with their line numbers), along with the name of the new file, on the command line. You can write as many lines as you like. The general format is:

:*line_number[,line_number]***w** *filename*

For example, to write the third line of the buffer to a line named **three**, type:

:3w three<**CR**>

vi reports the successful creation of your new file with the following information:

```
"three" [New file] 1 line, 20 characters
```

To write your current line to a file, you can use a . (period) as the line address:

:.w junk<CR>

A new file called **junk** is created. It will contain only the current line in the **vi** buffer.

You can also write a whole section of the buffer to a new file by specifying a range of lines. For example, to write lines 23 through 37 to a file, type the following:

:23,37w *newfile*<**CR**>

Finding the Line Number

To determine the number of a line, move the cursor to it and type : (colon). The colon will appear at the bottom of the screen. Type .= after it and press the **RETURN** key.

```
If you want to know the number
of this line, type :.=<CR>

:.=
```

As soon as you press the **RETURN** key, your command line will disappear from the bottom line and be replaced by the number of your current line in the buffer.

Also, a <**CTRL**>**-g** provides this information.

```
If you want to know the number
of this line, type in :.=<CR>

34
```

You can move the cursor to any line in the buffer by typing **:** and the line number. The following command line means to go to the *n*th line of the buffer.

:*n*<CR>

Also, <**shift**>**-g** produces the same results.

Deleting the Rest of the Buffer

One of the easiest ways to delete all the lines between the current line and the end of the buffer is by using the line editor command **d** with the special symbols for the current and last lines.

:.,$d<CR>

The **.** represents the current line; the **$** sign, the last line.

Adding a File to the Buffer

To add text from a file below a specific line in the editing buffer, use the **:r** (read) command. For example, to put the contents of a file called **data** into your current file, place the cursor on the line above the place where you want it to appear. Type:

:r data<CR>

You may also specify the line number instead of moving the cursor. For example, to insert the file **data** below line 56 of the buffer, type:

Do not be afraid to experiment; you can use the <**u**> command to undo **ex** commands, too.

Making Global Changes

One of the most powerful commands in **ex** is the global command. The global command is given here to help those users who are familiar with the line editor. Even if you are not familiar with a line editor, you may want to try the command on a test file.

For example, say you have several pages of text about the DNA molecule in which you refer to its structure as a helix. Now you want to change every occurrence of the word helix to double helix. The **ex** editor's global command allows you to do this with one command line. First, you need to understand a series of commands.

:g/*pattern/command*<**CR**>
For each line containing *pattern*, execute the **ex** command named *command*. For example, type: **:g/helix**<**CR**>. The line editor will print all lines that contain the pattern helix.

:s/*pattern/new_words/*<**CR**>
This is the substitute command. The line editor searches for the first instance of the characters *pattern* on the current line and changes them to *new_words.*

:s/*pattern/new_words/***g**<**CR**>
If you add the letter g after the last delimiter of this command line, **ex** will change every occurrence of *pattern* on the current line. If you do not, **ex** will change only the first occurrence.

:g/*helix***/s//***double helix***/g**<**CR**>
This command line searches for the word helix. Each time helix is found, the substitute command substitutes two words, double helix, for every instance of helix on that line. The delimiters after the **s** do not need to have helix typed in again. The command remembers the word from the delimiters after the global command **g**. This is a powerful command. For a more detailed explanation of global and substitution commands, see Chapter 5.

Table 6-13 summarizes the line editor commands available in **vi**.

Table 6-13. Summary of Line Editor Commands

Command	Function
:	Shows that the commands that follow are line editor commands.
:sh<CR>	Temporarily returns you to the shell to perform shell commands.
<^d>	Escapes the temporary shell and returns you to the current window of **vi** to continue editing.
:n<**CR**>	Goes to the *n*th line of the buffer.
:*x*,*y***w** *data*<**CR**>	Writes lines from the number *x* through the number *y* into a new file (*data*).
:$<CR>	Goes to the last line of the buffer.
:.,$d<CR>	Deletes all the lines in the buffer from the current line to the last line.
:r *shell.file*<**CR**>	Inserts the contents of *shell.file* after the current line of the buffer.
:s/*text*/*new_words*/<**CR**>	Replaces the first instance of the characters *text* on the current line with *new_words*.
:s/*text*/*new_words*/**g<CR>**	Replaces every occurrence of *text* on the current line with *new_words*.
:g/*text*/**s//***new_words*/**g<CR>**	Replaces every occurrence of *text* in the file with *new_words*.

Commands for Quitting vi

There are five basic command sequences to quit the **vi** editor. Commands that are preceded by a colon (:) are line editor commands.

<**ZZ**> or **:wq**<**CR**>	Write the contents of the **vi** buffer to the file currently being edited and quit **vi**.
:w *filename*<*CR*> **:q**<**CR**>	Write the temporary buffer to a new file named *filename* and quit **vi**.
:w! *filename*<*CR*> **:q**<**CR**>	Overwrite an existing file called *filename* with the contents of the buffer and quit **vi**.
:q!<**CR**>	Quit **vi** without writing the buffer to a file, and discard all changes made to the buffer.
:q<**CR**>	Quit **vi** without writing the buffer to a file. This works only if you have made no changes to the buffer; otherwise, **vi** warns you that you must either save the buffer or use the **:q!**<**CR**> command to terminate.

The <**ZZ**> command and **:wq** command sequence both write the contents of the buffer to a file, quit **vi**, and return you to the shell. You have tried the <**ZZ**> command. Now try to exit **vi** with **:wq**. **vi** remembers the name of the file currently being edited, so you do not have to specify it when you want to write the buffer's contents back into the file. Type:

:wq<**CR**>

The system responds in the same way it does for the <**ZZ**> command. It tells you the name of the file, and reports the number of lines and characters in the file.

What must you do to give the file a different name? For example, suppose you want to write to a new file called **junk**. Type:

:w junk<**CR**>

After you write to the new file, leave **vi**. Type:

:q<CR>

If you try to write to an existing file, you receive a warning. For example, if you try to write to a file called **johnson**, the system responds with:

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

If you want to replace the contents of the existing file with the contents of the buffer, use the **:w!** command to overwrite **johnson**:

:w! johnson<CR>

Your new file overwrites the existing one.

If you edit a file called **memo**, make some changes to it, and then decide you do not want to keep the changes, or if you accidentally press a key that gives **vi** a command you cannot undo, leave **vi** without writing to the file. Type:

:q!<CR>

Table 6-14 summarizes the quit commands.

Table 6-14. Summary of the Quit Commands

Command	Function
<ZZ>	Write the file and quit **vi**.
:wq<CR>	Write the file and quit **vi**.
:w *filename*<**CR**> **:q<CR>**	Write the editing buffer to a new file (*filename*) and quit **vi**.
:w! *filename*<**CR**> **:q<CR>**	Overwrite an existing file (*filename*) with the contents of the editing buffer and quit **vi**.
:q!<CR>	Quit **vi** without writing buffer to a file.
:q<CR>	Quit **vi** without writing the buffer to a file.

Special Options for vi

The **vi** command has some special options. It allows you to:

- recover a file lost by an interrupt to the operating system
- place several files in the editing buffer and edit each in sequence
- view the file at your own pace by using the **vi** cursor positioning commands

Recovering a File Lost by an Interrupt

If there is an interrupt or disconnect, the system exits the **vi** command without writing the text in the buffer back to its file. However, the operating system stores a copy of the buffer for you. When you log back into the operating system, you can restore the file with the **−r** option for the **vi** command. Type:

vi −r *filename*<**CR**>

The changes you made to *filename* before the interrupt occurred are now in the **vi** buffer. You can continue editing the file, or you can write the file and quit **vi**. The **vi** editor will remember the file name and write to that file.

Editing Multiple Files

If you want to edit more than one file in the same editing session, issue the **vi** command, specifying each file name. Type:

vi *file1file2*<**CR**>

vi responds by telling you how many files you are going to edit. For example:

```
2 files to edit
```

After you have edited the first file, write your changes (in the buffer) to the file (*file1*). Type:

:w<**CR**>

The system response to the **:w** <**CR**> command is a message at the bottom of the screen giving the name of the file, and the number of lines and characters in that file. Then you can bring the next file into the editing buffer by using the **:n** command. Type:

:n<**CR**>

The system responds by printing a notice at the bottom of the screen, telling you the name of the next file to be edited and the number of characters and lines in that file.

Select two of the files in your current directory. Then enter **vi** and place the two files in the editing buffer at the same time. Notice the system responses to your commands at the bottom of the screen.

Viewing a File

It is often convenient to be able to inspect a file by using **vi**s powerful search and scroll capabilities. However, you might want to protect yourself against accidentally changing a file during an editing session. The read-only option prevents you from writing in a file. To avoid accidental changes, you can set this option by invoking the editor as **view** rather than **vi**.

Table 6-15 summarizes the special options for **vi**.

Table 6-15. Summary of Special Options for **vi**

Option	Function
vi *file1file2file3*<**CR**>	Enter three files (*file1*, *file2*, and *file3*) into the **vi** buffer to be edited.
:w<**CR**> **:n**<**CR**>	Write the current file and call the next file into the buffer.
vi −r *file1*<**CR**>	Restore the changes made to *file1*.
VIEW*file*<**CR**>	Read file only.

Exercise 6

6-1. Try to restore a file lost by an interrupt.

Enter **vi**, create some text in a file called **exer6**. Turn off your terminal without writing to a file or leaving **vi**. Turn your terminal back on, and log in again. Then try to get back into **vi** and edit **exer6**.

6-2. Place **exer1** and **exer2** in the **vi** buffer to be edited. Write **exer1** and call in the next file in the buffer, **exer2**.

Write **exer2** to a file called **junk**.

Quit **vi**.

6-3. Try out the command:

vi exer*<CR>

What happens? Try to quit all the files as quickly as possible.

6-4. Look at **exer4** in read-only mode.

Scroll forward.

Scroll down.

Scroll backward.

Scroll up.

Quit and return to the shell.

Answers to Exercises

There is often more than one way to perform a task in **vi**. Any method that works is correct. The following are suggested ways of doing the exercises:

Exercise 1

1-1. Ask your System Administrator for your terminals system name. Type:

TERM=*terminal_name*<**CR**>
export TERM
TermSetup (This may be handled by your **.profile**.)

1-2. Enter the **vi** command for a file called **exer1**:

vi exer1<CR>

Then use the append command (**<a>**) to enter the following text in your file:

1-3. Use the **<k>** and **<h>** commands.

1-4. Use the **<x>** command.

1-5. Use the **<j>** and **<l>** commands.

1-6. Enter **vi** and use the append command (**<a>**) to enter the following text:

and byte by byte<ESC>

Then use **<j>** and **<l>** to move to the last line and character of the file. Use the **<a>** command again to add text. You can create a new line by pressing the RETURN key. To leave text input mode, press the ESCAPE key.

1-7. Type:

<ZZ>

1-8. Type:

vi exer1<CR>

Notice the system response:

```
"exer1" 7 lines, 102 characters
```

Exercise 2

2-1. Type:

vi exer2<CR>

<a>1<CR>
2<CR>
3<CR>
.
.
.
48<CR>
49<CR>
50<ESC>

2-2. Type:

<^f>
<^b>
<^u>
<^d>

Notice the line numbers as the screen changes.

2-3. Type:

<G>
<o>
123456789 123456789<ESC>
<7h>
<3l>

Typing **<7h>** puts the cursor
on the **2** in the second set of numbers.
Typing **<3l>** puts the cursor
on the **5** in the
second set of numbers.

2-4. **$** = end of line
0 = first character in the line

2-5. Type:

<^>
<w>
<b>
<e>

2-6. Type:

<1G>
<M>
<L>
<H>

2-7. Type:

/8
<n>
/48

Exercise 3

3-1. Type:

vi exer3<CR>

3-2. Type:

<a> Append text <CR>
Insert text<CR>
a computer's <CR>
job is boring.<ESC>

3-3. Type:

<O>
financial statement and<ESC>

3-4. Type:

<3G>
<i>Delete text<CR><ESC>

The text in your file now reads:

```
Append text
Insert text
Delete text
a computer's
financial statement and
job is boring.
```

3-5. The current line is `a computer's`. To create a line of text below that line use the **<o>** command.

3-6. The current line is `byte of the budget`.

<G> puts you on the bottom line.
<A> lets you begin appending at the end of the line.
<CR> creates the new line.

Add the sentence: **But, it is an exciting machine.**

<ESC> leaves append mode.

3-7. Type:

<1G>
/text
<i>some<space bar><ESC>

3-8. **<ZZ>** will write the buffer to **exer3** and return you to the shell.

Exercise 4

4-1. Type:

vi exer4<CR>
<a> When in the course of human events<CR>
there are many repetitive, boring<CR>
chores, then one ought to get a<CR>
robot to perform those chores.<ESC>

4-2. Type:

<2G>
<A> tedious and unsavory<8BACKSPACE><CR>
<ESC>

Press **<h>** until you get to the `b` of `boring`. Then type: **<dw>**. (You can also use **<6x>**.)

4-3. You are at the second line. Type:

<2j>
<I> congenial and computerized<ESC>
<dd>

To delete the line and leave it blank, type in:

<0> (zero moves the cursor to the beginning of the line)
<D>

<H>
<3dd>

4-4. Write and quit **vi**.

<ZZ>

Remove the file.

rm exer4<CR>

Exercise 5

5-1. Type:

```
vi exer2<CR>
<8G>
<cc> END OF FILE <ESC>
```

5-2. Type:

```
<1G>
<8"zyy>
<G>
<"zp>
```

5-3. Type:

```
<8G>
<cc> 8 is great<ESC>
```

5-4. Type:

```
<G>
<2w>
<cw>
EXERCISE<ESC>
<2b>
<cw>
TO<ESC>
```

Exercise 6

6-1. Type:

vi exer6<CR>
<a> (append several lines of text)
<ESC>

Turn off the terminal.

Turn on the terminal.
Log in on operating system. Type:

vi −r exer6<CR>
:wq<CR>

6-2. Type:

vi exer1 exer2<CR>
:w<CR>
:n<CR>

:w junk<CR>
<ZZ>

6-3. Type:

vi exer*<CR>

(Response:)
`8 files to edit` (**vi** calls all files with names that begin with **exe**

<ZZ>
<ZZ>

6-4. Type:

view exer4<CR>

<^f>
<^d>
<^b>
<^u>
:q<CR>

7 Shell Tutorial

Introduction

This chapter describes how to use the shell to do routine tasks. For example, it shows you how to use the shell to manage your files, to manipulate file contents, and to group commands together to make programs the shell can execute for you.

The chapter has two major sections. The first, *Shell Command Language*, covers using the shell as a command interpreter. It tells you how to use shell commands and characters with special meanings to manage files, redirect standard input and output, and execute and terminate processes. The second section, *Shell Programming*, covers using the shell as a programming language. It tells you how to create, execute, and debug programs made up of commands, variables, and programming constructs like loops and case statements. Finally, it tells you how to modify your login environment.

The chapter offers many examples. You should login to your operating system and recreate the examples as you read the text. As in the other examples in this guide, different type (**bold,** *italic,* and `constant width`) is used to distinguish your input from the operating system output. See *Notations and Conventions* in Chapter 1 for details.

In addition to the examples, there are exercises at the end of both sections. The exercises can help you better understand the topics discussed. The answers to the exercises are at the end of the chapter.

NOTE

> Your operating system may not have all commands referenced in this chapter. If you cannot access a command, check with your System Administrator.

If you want an overview of how the shell functions as both command interpreter and programming language, see Chapters 1 and 4 before reading this chapter. Also, refer to Appendix E, *Summary of Shell Command Language.*

Shell Command Language

This section introduces commands and, more importantly, some characters with special meanings that let you:

- find and manipulate a group of files by using pattern matching
- run a command in the background or at a specified time
- run a group of commands sequentially
- redirect standard input and output from and to files and other commands
- terminate processes

It first covers the characters having special meanings to the shell and then covers the commands for carrying out the tasks listed above. For your convenience, Table 7-1 summarizes the characters with special meanings that are discussed in this chapter.

Table 7-1. Characters with Special Meanings in the Shell Language

Character	Function
* ? [ ]	metacharacters that provide a shortcut for specifying file names by pattern matching
&	places commands in background mode, leaving your terminal free for other tasks
;	separates multiple commands on one command line
\	turns off the meaning of special characters such as *, ?, [ ], &, ;, >, <, and |.
'...'	single quotes turn off the delimiting meaning of a space and the special meaning of all special characters
"..."	double quotes turn off the delimiting meaning of a space and the special meaning of all special characters *except* $ and `
>	redirects output of a command into a file (replaces existing contents)
<	redirects command input so that it comes from a file
>>	redirects output of a command so that it is added to the end of an existing file
|	creates a pipe of the output of one command to the input of another command
`...`	grave accents allow the output of a command to be used directly as arguments on a command line
$	used with positional parameters and user-defined variables; also used as the default shell prompt symbol

Metacharacters

Metacharacters, a subset of the special characters, represent other characters. They are sometimes called *wild cards*, because they are like the joker in card games that can be used for any card. The metacharacters * (asterisk), **?** (question mark), and **[ ]** (brackets) are discussed here.

These characters are used to match file names or parts of file names, thereby simplifying the task of specifying files or groups of files as command arguments. (The files whose names match the patterns formed from these metacharacters must already exist.) This is known as *file-name expansion.* For example, you may want to refer to all file names containing the letter "a", all file names consisting of five letters.

Matching All Characters: Asterisk (*)

The asterisk (*) matches any string of characters including a null (empty) string. You can use the * to specify a full or partial file name. The * alone refers to all the file and directory names in the current directory. To see the effect of the *, try it as an argument to the **echo**(1) command. Type:

echo *<CR>

The **echo** command displays its arguments on your screen. Notice that the system response to **echo** * is a listing of all file names in your current directory. However, the file names are displayed horizontally instead of in vertical columns such as those produced by the **ls** command.

Table 7-2 summarizes the syntax and capabilities of the **echo** command.

> CAUTION
>
> **The * is a powerful character. For example, if you type rm * you will erase all the files in your current directory.**

Table 7-2. Summary of the **echo** Command

Command Recap		
echo – write any arguments to the output		
Command	**Options**	**Arguments**
echo	none	any character(s)
Description:	**echo** writes arguments that are separated by blanks and ended with <**CR**> to the output.	
Remarks:	In shell programming, **echo** is used to issue instructions, to redirect words or data into a file, and to pipe data into a command. All these uses are discussed later in this chapter.	

For another example, you have written several reports and named them **report**, **report1**, **report1a**, **report1b.01**, **report25**, and **report316**. By typing **report1*** you can refer to all files that are part of report1, collectively. To find out how many reports you have written, you can use the **ls** command to list all files that begin with the string "report," as shown in the following example.

```
$ ls report*<CR>
report
report1
report1a
report1b.01
report25
report316
$
```

The * matches any characters after the string "report," including no letters at all. Notice that * matches the files in numerical and alphabetical order. A quick and easy way to print the contents of your report files in order on your screen is by typing the following command:

pr report*<CR>

Now try another exercise. Choose a character that all file names in your current directory have in common, e.g., a lowercase "a." Then request a listing of those files by referring to that character. For example, if you choose a lowercase "a," type the following command line:

ls *a*<CR>

The system responds by printing the names of all the files in your current directory that contain a lowercase "a."

The * can represent characters in any part of the file name. For example, if you know that several files have their first and last letters in common, you can request a list of them on that basis. For such a request, your command line might look like this:

ls F*E<CR>

The system response is a list of file names that begin with F, end with E, and are in the following order:

F123E
FATE
FE
Fig3.4E

The order is determined by the ASCII sort sequence: (1) numbers; (2) uppercase letters; (3) lowercase letters.

Matching One Character: Question Mark (?)

The question mark (**?**) matches any single character of a file name. For example, you have written several chapters in a book that has 12 chapters, and you want a list of those you have finished through Chapter 9. Use the **ls** command with the **?** to list all chapters that begin with the string "chapter" and end with any single character:

```
$ ls chapter?<CR>
chapter1
chapter2
chapter5
chapter9
$
```

The system responds by printing a list of all file names that match.

Although **?** matches any one character, you can use it more than once in a file name. To list the rest of the chapters in your book, type:

ls chapter??<CR>

If you want to list all the chapters in the current directory, use the *:

ls chapter*

Using the * or ? To Correct Typing Errors

Suppose you use the **mv**(1) command to move a file, and you make an error and enter a character in the file name that is not printed on your screen. The system incorporates this non-printing character into the name of your file and subsequently requires it as part of the file name. If you do not include this character when you enter the file name on a command line, you get an error message. You can use * or **?** to match the file name with the non-printing character and rename it to the correct name.

Try the following example:

1. Make a very short file called **trial**.
2. Type: **mv trial trial<^g>1<CR>**

   (Remember, to type **<^g>** you must hold down the CONTROL key and press the **<g>** key.)
3. Type: **ls trial1<CR>**

   The system responds with an error message:

   ```
   $ ls trial1<CR>
   trial1: no such file or directory
   $
   ```
4. Type: **ls trial?1<CR>**

   The system responds with the file name **trial1** (including the non-printing character), verifying that this file exists. Use the **?** again to correct the file name.

   ```
   $ mv trial?1 trial1<CR>
   $ ls trial1<CR>
   trial1
   $
   ```

Matching One of a Set: Brackets ([ ])

Use brackets ([ ]) when you want the shell to match any one of several possible characters that may appear in one position in the file name. For example, if you include **[crf]** as part of a file name pattern, the shell will look for file names that have the letter "c", the letter "r", or the letter "f" in the specified position, as the following example shows:

```
$ ls [crf]at<CR>
cat
fat
rat
$
```

This command displays all file names that begin with the letter "c", "r", or "f" and end with the letters "at". Characters that can be grouped within brackets in this way are collectively called a *character class*.

Brackets can also be used to specify a range of characters, whether numbers or letters. For example, the following specifies that the shell matches any files named **chapter1** through **chapter5**.

chapter[1-5]

This is an easy way to handle only a few chapters at a time.

Try the **pr** command with an argument in brackets:

$ pr chapter[2-4]<CR>

This command prints the contents of **chapter2**, **chapter3**, and **chapter4**, in that order, on your terminal.

A character class may also specify a range of letters. If you specify **[A-Z]**, the shell looks only for uppercase letters; if **[a-z]**, only lowercase letters.

The uses of the metacharacters are summarized in Table 7-3. Try out the metacharacters on the files in your current directory.

Table 7-3. Summary of Metacharacters

Notation	Function
*	matches any string of characters, including an empty (null) string
?	matches any single character
[*chars*]	matches one of the sequence of characters specified within the brackets
[*char* −*char*]	matches one of the range of characters specified

Special Characters

The shell language has other special characters that perform a variety of useful functions. Some of these additional special characters are discussed in this section; others are described in the next section, *Input and Output Redirection*.

Executing in Background: Ampersand (&)

Some shell commands take considerable time to execute. The ampersand (**&**) is used to execute commands in background mode, freeing your terminal for other tasks. The general format for running a command in background mode is:

command **&<CR>**

NOTE

You should not run interactive shell commands, for example **read** (see *Using the read Command* in this chapter), in the background.

In the example below, the shell is performing a long search in background mode. Specifically, the **grep**(1) command is searching for the string "delinquent" in the file **accounts**. Notice the **&** is the last character of the command line:

```
$ grep delinquent accounts &<CR>
21940
$
```

When you run a command in the background, the operating system outputs a process number; **21940** is the process number in the example. You can use this number to stop the execution of a background command. (Stopping the execution of processes is discussed in the *Executing and Terminating Processes* section.) The prompt on the last line means the terminal is free and waiting for your commands; **grep** has started running in background.

Running a command in background affects only the availability of your terminal; it does not affect the output of the command. Whether or not a command is run in background, it prints its output on your terminal screen, unless you redirect it to a file. (See *Redirecting Output*, later in this chapter, for details.)

If you want a command to continue running in background after you log off, you can submit it with the **nohup**(1) command. (This is discussed in *Using the nohup Command*, later in this chapter.)

Executing Commands Sequentially: Semicolon (;)

You can type two or more commands on one line as long as each pair is separated by a semicolon (;):

command1; *command2*; *command3*<**CR**>

The operating system executes the commands in the order that they appear in the line and prints all output on the screen. This process is called *sequential execution*.

Try this exercise to see how the ; works. First, type:

```
cd; pwd; ls<CR>
```

The shell executes these commands sequentially:

1. **cd** changes your location to your login directory
2. **pwd** prints the full path name of your current directory
3. **ls** lists the files in your current directory

If you do not want the systems responses to these commands to appear on your screen, refer to *Redirecting Output* for instructions.

Turning Off Special Meanings: Backslash

The shell interprets the backslash (\) as an escape character that allows you to turn off any special meaning of the character immediately after it. To see how this works, try the following exercise. Create a two-line file called **trial** that contains the following text:

```
The all * game
was held in Summit.
```

Use the **grep** command to search for the asterisk in the file, as shown in the following example:

```
$ grep \* trial<CR>
The all * game
$
```

The **grep** command finds the * in the text and displays the line in which it appears. Without the \, the * would be a metacharacter to the shell and would match all file names in the current directory.

Turning Off Special Meanings: Quotes

Another way to escape the meaning of a special character is to use quotation marks. Single quotes ('...') turn off the special meaning of any character. Double quotes ("...") turn off the special meaning of all characters except **$** and ` (grave accent) retain their special meanings within double quotes. An advantage of using quotes is that numerous special characters can be enclosed in the quotes; this can be more concise than using the backslash.

For example, if your file named **trial** also contained the following line:

```
He really wondered why? Why???
```

You could use the **grep** command to match the line with the three question marks:

```
$ grep '???' trial<CR>
He really wondered why? Why???
$
```

If you had instead entered the command:

```
grep ??? trial<CR>
```

The three question marks would have been used as shell metacharacters and matched all file names of length three.

Using Quotes to Turn Off the Meaning of a Space

A common use of quotes as escape characters is for turning off the special meaning of the blank space. The shell interprets a space on a command line as a delimiter between the arguments of a command. Both single and double quotes allow you to escape that meaning.

For example, to locate two or more words that appear together in text, make the words a single argument (to the **grep** command) by enclosing them in quotes. To find the two words "The all" in your file **trial**, enter the following command line:

```
$ grep 'The all' trial<CR>
The all * game
$
```

grep finds the string "The all" and prints the line that contains it. What would happen if you did not put quotes around that string?

The ability to escape the special meaning of a space is helpful when you are using the **banner**(1) command. This command prints a message across a terminal screen in large, poster size letters.

To execute **banner**, specify a message consisting of one or more arguments (in this case usually words), separated on the command line by spaces. The **banner** uses these spaces to delimit the arguments and print each argument on a separate line.

To print more than one argument on the same line, enclose the words, together, in double quotes. For example, to send a birthday greeting to another user, type:

banner happy birthday to you<CR>

The command prints your message as a four-line banner. Now print the same message as a three-line banner. Type:

banner happy birthday "to you"<CR>

Notice that the words "to" and "you" now appear on the same line. The space between them has lost its meaning as a delimiter.

Table 7-4 summarizes the syntax and capabilities of the **banner** command.

Table 7-4. Summary of the **banner** Command

Command Recap banner – make posters		
Command	**Options**	**Arguments**
banner	none	characters
Description:	**banner** displays up to ten characters in large letters	
Remarks:	Later in this chapter you will learn how to redirect the **banner** command into a file to be used as a poster.	

Input and Output Redirection

In the operating system, some commands expect to receive their input from the keyboard (standard input) and most commands display their output at the terminal (standard output). However, the operating system lets you reassign the standard input and output to other files and programs. This is known as redirection. With redirection, you can tell the shell to:

- take its input from a file instead of the keyboard
- send its output to file instead of the terminal
- use a program as the source of data for another program

You use a set of operators, the less than sign (<), the greater than sign (>), two greater than signs (>>), and the pipe (|) to redirect input and output.

Redirecting Input: < Sign

To redirect input, specify a file name after a less than sign (<) on a command line:

command < *file*<**CR**>

For example, assume that you want use the **mail**(1) command (described in Chapter 8) to send a message to another user with the login **colleague** and that you already have the message in a file named **report**. You can avoid retyping the message by specifying the file name as the source of input:

mail colleague < **report**<**CR**>

Redirecting Output to a File: > Sign

To redirect output, specify a file name after the greater than sign (>) on a command line:

command > *file*<**CR**>

CAUTION

If you redirect output to a file that already exists, the output of your command will overwrite the contents of the existing file.

Before redirecting the output of a command to a particular file, make sure that a file by that name does not already exist, unless you do not mind losing it. Because the shell does not allow you to have two files of the same name in a directory, it overwrites the contents of the existing file with the output of your command if you redirect the output to a file with the existing files name. The shell does not warn you about overwriting the original file.

To make sure there is no file with the name you plan to use, run the **ls** command, specifying your proposed file name as an argument. If a file with that name exists, **ls** will list it; if not, you receive a message that the file was not found in the current directory. For example, checking for the existence of the files **temp** and **junk** would give you the following output:

```
$ ls temp<CR>
temp
$ ls junk<CR>
junk:  no such file or directory
$
```

This means you can name your new output file **junk**, but you cannot name it **temp** unless you no longer want the contents of the existing **temp** file.

Appending Output to an Existing File: >> Symbol

To keep from destroying an existing file, you can also use the double redirection symbol (>>), as follows:

command >> *file*<**CR**>

This appends the output of a command to the end of the file *file*. If *file* does not exist, it is created when you use the >> symbol this way.

The following example shows how to append the output of the **cat** command to an existing file. First, the **cat** command is executed on both files without output redirection to show their respective contents. Then the contents of **trial2** are added after the last line of **trial1** by executing the **cat** command on **trial2** and redirecting the output to **trial1**.

```
$ cat trial1<CR>
This is the first line of trial1.
Hello.
This is the last line of trial1.
$
$ cat trial2<CR>
This is the beginning of trial2.
Hello.
This is the end of trial2.
$
$ cat trial2 >> trial1<CR>
$ cat trial1<CR>
This is the first line of trial1.
Hello.
This is the last line of trial1.
This is the beginning of trial2.
Hello.
This is the end of trial2.
$
```

Useful Applications of Output Redirection

Redirecting output is useful when you do not want it to appear on your screen immediately or when you want to save it. Output redirection is also especially useful when you run commands that perform clerical chores on text files. Two such commands are **spell** and **sort**.

The spell Command

The **spell** program compares every word in a file against its internal vocabulary list and prints a list of all potential misspellings on the screen. If **spell** does not have a listing for a word (e.g., a persons name), it also reports that as a misspelling.

Running **spell** on a lengthy text file can take a long time and may produce a list of misspellings that is too long to fit on your screen. **spell** prints all its output at once; if it does not fit on the screen, the command scrolls it continuously off the top until it has all been displayed. A long list of misspellings rolls off your screen quickly and may be difficult to read.

You can avoid this problem by redirecting the output of **spell** to a file. In the following example, **spell** searches a file named **memo** and places a list of misspelled words in a file named **misspell**:

$ spell memo > misspell<CR>

Table 7-5 summarizes the syntax and capabilities of the **spell** command.

Table 7-5. Summary of the **spell** Command

Command Recap spell – find spelling errors		
Command	**Options**	**Arguments**
spell	available*	*file*
Description:	**spell** collects words from a specified file or files and looks them up in a spelling list. Words that are not on the spelling list are displayed on your terminal.	
Options:	**spell** has several options, including one for checking British spellings.	
Remarks:	The list of misspelled words can be redirected into a file.	

* See the **spell**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

The sort Command

The **sort** command arranges the lines of a specified file in alphabetical order (see Chapter 3 for details). Because users generally want to keep a file that has been alphabetized, output redirection greatly enhances the value of this command.

Be careful to choose a new name for the file that will receive the output of the **sort** command (the alphabetized list). When **sort** is executed, the shell first empties the file that will accept the redirected output. Then it performs the sort and places the output in the blank file. If you type the following, the shell will empty **list** and then sort nothing into **list**.

sort list > list<CR>

Combining Background Mode and Output Redirection

Running a command in background does not affect the commands output; unless it is redirected, output is always printed on the terminal screen. If you are using your terminal to perform other tasks while a command runs in background, you are interrupted when the command displays its output on your screen. However, if you redirect that output to a file, you can work undisturbed.

For example, in the *Special Characters* section you learned how to execute the **grep** command in background with **&**. Now suppose you want to find occurrences of the word "test" in a file named **schedule**. Run the **grep** command in background and redirect its output to a file called **testfile**:

$ grep test schedule > testfile &<CR>

You can then use your terminal for other work and examine **testfile** when you have finished it.

Redirecting Output to a Command: Pipe (|)

The | character is called a *pipe*. Pipes are powerful tools that allow you to take the output of one command and use it as input for another command without creating temporary files. A multiple command line created in this way is called a *pipeline*.

The general format for a pipeline is:

command1 | *command2* | *command3*...<**CR**>

The output of *command1* is used as the input of *command2*. The output of *command2* is then used as the input for *command3*.

To understand the efficiency and power of a pipeline, consider the contrast between two methods that achieve the same results:

- To use the input/output redirection method, run one command and redirect its output to a temporary file. Then run a second command that takes the contents of the temporary file as its input. Finally, remove the temporary file after the second command has finished running.
- To use the pipeline method, run one command and pipe its output directly into a second command.

For example, you want to mail a happy birthday message in a banner to the owner of the login **david**. Doing this without a pipeline is a three-step procedure. You must:

1. Enter the **banner** command and redirect its output to a temporary file:

   banner happy birthday > message.tmp

2. Enter the **mail** command using **message.tmp** as its input:

   mail david < message.tmp

3. Remove the temporary file:

   rm message.tmp

However, by using a pipeline you can do this in one step:

banner happy birthday | mail david<**CR**>

A Pipeline Using the cut and date Commands

The **cut** and **date** commands provide a good example of how pipelines can increase the versatility of individual commands. The **cut** command allows you to extract part of each line in a file. It looks for characters in a specified part of the line and prints them. To specify a position in a line, use the **−c** option and identify the part of the file you want by the numbers of the spaces it occupies on the line.

For example, you want to display only the dates from a file called **birthdays**. The file contains the following list:

```
Anne      12/26
Klaus     7/4
Mary      10/18
Peter     11/9
Nandy     4/23
Sam       8/12
```

The birthdays appear between the ninth and thirteenth spaces on each line. To display them, type:

cut −c9-13 birthdays<CR>

The output is shown below:

```
12/26
7/4
10/18
11/9
4/23
8/12
```

Table 7-6 summarizes the syntax and capabilities of the **cut** command.

Table 7-6. Summary of the **cut** Command

Command Recap **cut** – cut out selected fields from each line of a file		
Command	**Options**	**Arguments**
cut	**−c**list **−f**list [**−d**]	*file*
Description:	**cut** extracts columns from a table or fields from each line of a file.	
Options:	**−c** lists the number of character positions from the left. A range of numbers such as characters 1–9 can be specified by **−c1–9.** **−f** lists the field number from the left separated by a delimiter described by **−d.** **−d** gives the field delimiter for **−f.** The default is a space. If the delimiter is a colon, this would be specified by **−d :** .	
Remarks:	If you find the **cut** command useful, you may also want to use the **paste** and **split** commands.	

The **cut** command is usually executed on a file. However, piping makes it possible to run this command on the output of other commands, too. This is useful if you want only part of the information generated by another command.

For example, you may want to have the time printed. The **date** command prints the day of the week, date, and time, as follows:

```
$ date<CR>
Sat Dec 27 13:12:32 EST 1986
```

Notice that the time is given between the twelfth and nineteenth spaces of the line. You can display the time (without the date) by piping the output of **date** into **cut**, specifying spaces **12–19** with the **–c** option. Your command line and its output will look like this:

```
$ date | cut –c12-19<CR>
13:14:56
```

Table 7-7 summarizes the syntax and capabilities of the **date** command.

Table 7-7. Summary of the **date** Command

Command Recap **date** - display the date and time		
Command	**Options**	**Arguments**
date	**+%m%d%y*** **+%H%%M%S**	available*
Description:	**date** displays the current date and time on your terminal	
Options:	**+%** followed by **m** (for month), **d** (for day), **y** (for year), **H** (for hour), **M** (for month), and **S** (for second) will echo these back to your terminal. You can add explanations, for example: **date ´+%H:%M is the time´**	
Remarks:	If you are working on a small computer system of which you are both a user and the System Administrator, you may be allowed to set the date and time using optional arguments to the **date** command. Check your reference manual for details. When working in a multiuser environment, the arguments are available only to the System Administrator.	

* See the **date**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Substituting Output for an Argument

The output of any command may be captured and used as arguments on a command line. This is done by enclosing the command in grave accents (`...`) and placing it on the command line in the position where the output should be treated as arguments. This is known as *command substitution.*

For example, you can substitute the output of the **date** and **cut** pipeline command used previously for the argument in a **banner** printout by typing the following command line:

$ banner `date | cut –c12–19`<CR>

Notice the results: the system prints a banner with the current time.

The *Shell Programming* section in this chapter shows you how you can also use the output of a command line as the value of a variable.

Executing and Terminating Processes

This section discusses the following topics:

- how to schedule commands to run at a later time by using the **batch** or **at** command
- how to obtain the status of active processes
- how to terminate active processes
- how to keep background processes running after you have logged off

Executing at a Later Time: batch and at Commands

The **batch** and **at** commands allow you to specify a command or sequence of commands to be run at a later time. With the **batch** command, the system determines when the commands run; with the **at** command, you determine when the commands run. Both commands expect input from standard input (the terminal); the list of commands entered as input from the terminal must be ended by pressing <^**d**> (CTRL-**d**).

The **batch** command is useful if you are running a process or shell program that uses a large amount of system time. The **batch** command submits a batch job (containing the commands to be executed) to the system. The job is put in a queue, and runs when the system load falls to an acceptable level. This frees the system to respond rapidly to other input and is a courtesy to other users.

The general format for **batch** is:

batch<CR>
first command<**CR**>
.
.
.
last command<**CR**>
<**^d**>

If there is only one command to be run with **batch**, you can enter it as follows:

batch *command_line*<**CR**>
<**^d**>

The next example uses **batch** to execute the **grep** command at a convenient time. Here **grep** searches all files in the current directory and redirects the output to the file **dol.file**.

```
$ batch grep dollar * > dol-file<CR>
<^d>
job 155223141.b at Sun Dec 7 11:14:54 1986
$
```

After you submit a job with **batch**, the system responds with a job number, date, and time. This job number is not the same as the process number that the system generates when you run a command in the background.

Table 7-8 summarizes the syntax and capabilities of the **batch** Command.

Table 7-8. Summary of the **batch** Command

Command Recap **batch –** execute commands at a later time		
Command	**Options**	**Input**
batch	none	*command_lines*
Description:	**batch** submits a batch job, which is placed in a queue and executed when the load on the system falls to an acceptable level.	
Remarks:	The list of commands must end with a <^**d**> (CTRL-**d**).	

The **at** command allows you to specify an exact time to execute the commands. The general format for the **at** command is:

at *time*<**CR**>
first command<**CR**>
.
.
.
last command<**CR**>
<^**d**>

The *time* argument consists of the time of day and, if the date is not today, the date.

The following example shows how to use the **at** command to mail a happy birthday banner to login **emily** on her birthday:

```
$ at 8:15am Feb 27<CR>
banner happy birthday | mail emily<CR>
<^d>
job 453400603.a at Thurs Feb 27 08:15:00 1986
$
```

Notice that the **at** command, like **batch**, responds with the job number, date, and time.

If you decide you do not want to execute the commands currently waiting in a **batch** or **at** job queue, you can erase those jobs by using the **−r** option of the **at** command with the job number. The general format is:

at −r *jobnumber*<**CR**>

Try erasing the previous **at** job for the happy birthday banner. Type in:

at −r 453400603.a<**CR**>

If you have forgotten the job number, the **at** **−l** command will give you a list of the current jobs in the **batch** or **at** queue, as the following screen shows (if you are logged in as **root**, the notation "**users = username**" also appears):

```
$ at -l<CR>
user = mylogin 168302040.a at Sat Nov 29 13:00:00 1986
user = mylogin 453400603.a at Fri Feb 27 08:15:00 1987
$"
```

Notice that the system displays the job number and the time the job will run.

Using the **at** command, mail yourself the file **memo** at noon, to tell you it is lunch time. (You must redirect the file into **mail** unless you use the **here** document, described in the *Shell Programming* section.) Then try the **at** command with the **−l** option:

```
$ at 12:00pm<CR>
mail mylogin < memo<CR>
<^d>
job 263131754.a at Jun 30 12:00:00 1986
$
$ at -l<CR>
user = mylogin 263131754.a at Jun 30 12:00:00 1986
$
```

Table 7-9 summarizes the syntax and capabilities of the **at** command.

Table 7-9. Summary of the **at** Command

Command Recap **at** – execute commands at a specified time		
Command	**Options**	**Arguments**
at	**–r** **–l**	*time (date)* *jobnumber*
Description:	Executes commands at the time specified. You can use between one and four digits, and am or pm to show the time. To specify the date, give a month name followed by the number for the day. You do not need to enter a date if you want your job to run the same day. See the **at**(1) manual page in the *User's Reference Manual* for other default times.	
Options:	The **–r** option with the job number removes previously scheduled jobs. The **–l** option (no arguments) reports the job number and status of all scheduled **at** and **batch** jobs.	
Remarks:	Examples of how to specify times and dates with the **at** command: **at 08:15am Feb 27** **at 5:14pm Sept 24**	

Obtaining the Status of Running Processes

The **ps** command gives you the status of all the processes you are currently running. For example, you can use the **ps** command to show the status of all processes that you run in the background using **&** (described in the earlier section *Special Characters*).

The next section, *Terminating Active Processes*, discusses how you can use the PID (process identification) number to stop a command from executing. A PID is a number from 1 to 30,000 that the operating system assigns to each active process.

In the following example, **grep** is run in the background, then the **ps** command is issued. The system responds with the process identification (PID) and the terminal identification (TTY) number. It also gives the cumulative execution time for each process (TIME), and the name of the command that is being executed (COMMAND).

```
$ grep word * > temp &<CR>
28223
$
$ ps<CR>
PID          TTY  TIME COMMAND
28124        tty100:00   sh
28223        tty100:04   grep
28224        tty100:04   ps
$
```

Notice that the system reports a PID number for the **grep** command, as well as for the other processes that are running: the **ps** command itself, and the **sh** (shell) command that runs while you are logged in. The shell program **sh** interprets the shell commands and is discussed in Chapters 1 and 4.

Table 7-10 summarizes the syntax and capabilities of the **ps** command.

Table 7-10. Summary of the **ps** Command

Command Recap **ps** – report process status		
Command	**Options**	**Arguments**
ps	several*	none
Description:	**ps** displays information about active processes.	
Options:	Several. If none are specified, **ps** displays the status of all active processes you are running.	
Remarks:	Gives you the PID (process ID). This is needed to **kill** a process (stop the process from executing).	

* See the **ps**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Terminating Active Processes

The **kill** command is used to terminate active shell processes. The general format for the **kill** command is:

kill *PID*<**CR**>

You can use the **kill** command to terminate processes that are running in background. Note that you cannot terminate background processes by pressing the BREAK or CINTR key.

The following example shows how you can terminate the **grep** command that you started executing in background in the previous example:

```
$ kill 28223<CR>
28223 Terminated
$
```

Notice the system responds with a message and a **$** prompt, showing that the process has been killed. If the system cannot find the PID number you specify, it responds with an error message:

```
kill:28223:No such process
```

Table 7-11 summarizes the syntax and capabilities of the **kill** command.

Table 7-11. Summary of the **kill** Command

Command Recap **kill** – terminate a process		
Command	**Options**	**Arguments**
kill	available*	*job number* or *PID*
Description:	**kill** terminates the process specified by the PID number.	

* See the **kill**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Using the nohup Command

All processes are killed when you log off. If you want a background process to continue running after you log off, you must use the **nohup** command to submit that background command.

To execute the **nohup** command, follow this format:

nohup *command* **&**<**CR**>

Notice that you place the **nohup** command before the command you intend to run as a background process.

For example, you want the **grep** command to search all files in the current directory for the string "word" and redirect the output to a file called **word.list**, and you wish to log off immediately afterward. Type the command line as follows:

nohup grep word * > word.list & <CR>

You can terminate the **nohup** command by using the **kill** command.

Table 7-12 summarizes the syntax and capabilities of the **nohup** command.

Table 7-12. Summary of the **nohup** Command

Command Recap **nohup** – prevents interruption of command execution by hang ups		
Command	**Options**	**Arguments**
nohup	*none*	*command line*
Description:	Executes a command line, even if you hang up or quit the system.	

Now that you have mastered these basic shell commands and notations, use them in your shell programs. The exercises that follow will help you practice using shell command language. The answers to the exercises are at the end of the chapter.

Command Language Exercises

1-1. What happens if you use an * (asterisk) at the beginning of a file name? Try to list some of the files in a directory using the * with the last letter of one of your file names. What happens?

1-2. Try the following two commands; enter them as follows:

cat[0-9]*<CR>
echo *<CR>

1-3. Is it acceptable to use a **?** at the beginning or in the middle of a file name generation? Try it.

1-4. Do you have any files that begin with a number? Can you list them without listing the other files in your directory? Can you list only those files that begin with a lowercase letter between a and m? (Hint: use a range of numbers or letters in **[ ]**).

1-5. Is it acceptable to place a command in background mode on a line that is executing several other commands sequentially? Try it. What happens? (Hint: use **;** and **&.**) Can the command in background mode be placed in any position on the command line? Try placing it in various positions. Experiment with each new character that you learn to see the full power of the character.

1-6. Redirect the output of **pwd** and **ls** into a file by using the following command line:

cd; pwd; ls; ed trial<CR>

Remember, if you want to redirect both commands to the same file, you have to use the >> (append) sign for the second redirection. If you do not, you will wipe out the information from the **pwd** command.

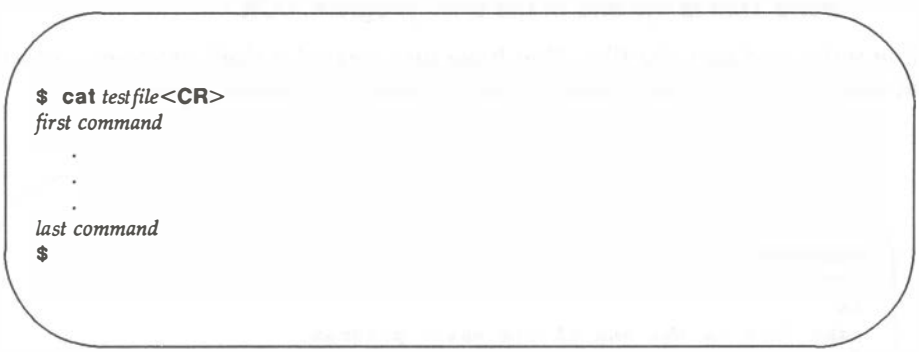
1-7. Instead of cutting the time out of the **date** response, try redirecting only the date, without the time, into **banner**. What is the only part you need to change in the time command line?

banner 'date | cut –c12-19'<CR>

Shell Programming

You can use the shell to create programs and/or new commands. Such programs are also called *shell procedures*. This section tells you how to create and execute shell programs using commands, variables, positional parameters, return codes, and basic programming control structures.

The examples of shell programs in this section are shown two ways. First, the **cat** command is used in a screen to display the contents of a file containing a shell program:



Second, the results of executing the shell program appear after a command line:

$ *testfile*<**CR**>
program_output
$

You should be familiar with an editor before you try to create shell programs. See to the tutorials in Chapter 5 (for the **ed** editor) and Chapter 6 (for the **vi** editor).

Shell Programs

Creating a Simple Shell Program

Create a simple shell program that does the following tasks, in order:

1. Print the current directory.
2. List the contents of that directory.
3. Display this message on your terminal: "This is the end of the shell program."

Create a file called **dl** (short for directory list) using your editor of choice, and enter the following:

pwd<CR>
ls<CR>
echo This is the end of the shell program.<CR>

Now write and quit the file. You have just created a shell program! You can **cat** the file to display its contents, as the following screen shows:

```
$ cat dl<CR>
pwd
ls
echo This is the end of the shell program.
$
```

Executing a Shell Program

One way to execute a shell program is to use the **sh** command. Type:

sh dl<CR>

The **dl** command is executed by **sh**, and the path name of the current directory is printed first, then the list of files in the current directory, and finally, the comment `This is the end of the shell program`. The **sh** command provides a good way to test your shell program to make sure it works.

If **dl** is a useful command, you can use the **chmod** command to make it an executable file; then you can type **dl** by itself to execute the command it contains. The following example shows how to use the **chmod** command to make a file executable and then run the **ls −l** command to verify the changes you have made in the permissions:

```
$ chmod u+x dl<CR>
$ ls -l<CR>
total 2
-rw-------      1  login login  3661  Nov  2   10:28 mbox
-rwx------      1  login login    48  Nov 15   10:50 dl
$
```

Notice that **chmod** turns on permission to execute (**+x**) for the user (**u**). Now **dl** is an executable program. Try to execute it. Type:

dl<CR>

You get the same results as before, when you entered **sh dl** to execute it. For further details about the **chmod** command, see Chapter 3.

Creating a bin Directory for Executable Files

To make your shell programs accessible from all your directories, you can make a **bin** directory from your login directory and move the shell files to your **bin**.

You must also set your shell variable **PATH** to include your **bin** directory:

```
PATH=$PATH:$HOME/bin
```

See *Variables* and *Using Shell Variables* in this chapter for more information about **PATH**.

The following example will remind you which commands are necessary. In this example, **dl** is in the login directory. Type these command lines:

```
cd<CR>
mkdir bin<CR>
mv dl bin/dl<CR>
```

Move to the **bin** directory and type the **ls –l** command. Does **dl** still have execute permission?

Now move to a directory other than the login directory, and type the following command:

```
dl<CR>
```

What happened?

Table 7-13 summarizes your new shell program, **dl**.

Table 7-13. Summary of the **dl** Shell Program

Shell Program Recap **dl** – display the directory path and directory contents (user defined)	
Command	**Arguments**
dl	none
Description:	**dl** displays the output of the shell command **pwd** and **ls**.

It is possible to give the **bin** directory another name; if you do so, you need to change your shell variable **PATH** again.

Warnings About Naming Shell Programs

You can give your shell program any appropriate file name. However, you should not give your program the same name as a system command. If you do, the system executes your command instead of the system command. For example, if you had named your **dl** program **mv**, each time you tried to move a file, the system would have executed your directory list program instead of **mv**.

Another problem can occur if you name the **dl** file **ls**, and then try to execute the file. You would create an infinite loop, since your program executes the **ls** command. After some time, the system would give you the following error message:

```
Too many processes, cannot fork
```

What happened? You typed in your new command, **ls**. The shell read and executed the **pwd** command. Then it read the **ls** command in your program and tried to execute your **ls** command. This formed an infinite loop.

SYSTEM V/88 designers wisely set a limit on how many times an infinite loop can execute. One way to keep this from happening is to give the path name for the system's **ls** command, **/bin/ls**, when you write your own shell program.

The following **ls** shell program would work:

```
$ cat ls<CR>
pwd
/bin/ls
echo This is the end of the shell program
```

If you name your command **ls**, then you can only execute the system **ls** command by using its full path name, **/bin/ls**.

Variables

Variables are the basic data objects shell programs manipulate, other than files. Here we discuss three types of variables and how you can use them:

- positional parameters
- special parameters
- named variables

Positional Parameters

A positional parameter is a variable within a shell program whose value is set from an argument specified on the command line invoking the program. Positional parameters are numbered and are referred to with a preceding **$**: e.g., **$1**, **$2**, **$3**.

A shell program may reference up to nine positional parameters. If a shell program is invoked with the followiwng command line, then positional parameter **$1** within the program is assigned the value **pp1**, positional parameter **$2** within the program is assigned the value **pp2**, and so on, when the shell program is invoked.

shell.prog pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9<CR>

Create a file called **pp** (short for positional parameters) to practice positional parameter substitution. Then enter the **echo** commands shown in the following screen. Enter the command lines so that running the **cat** command on your completed file will produce the following output:

```
$ cat pp<CR>
echo  The first positional parameter is: $1<CR>
echo  The second positional parameter is: $2<CR>
echo  The third positional parameter is: $3<CR>
echo  The fourth positional parameter is: $4<CR>
$
```

If you execute this shell program with the arguments **one**, **two**, **three**, and **four**, you will obtain the following results (first you must make the shell program **pp** executable using the **chmod** command):

```
$ chmod u+x pp<CR>
$
$ pp one two three four<CR>
The first positional parameter is: one
The second positional parameter is: two
The third positional parameter is: three
The fourth positional parameter is: four
$
```

The following screen shows the shell program **bbday**, which mails a greeting to the login entered in the command line:

```
$ cat bbday<CR>
banner happy birthday | mail $1
```

Try sending yourself a birthday greeting. If your login name is **sue**, your command line will be:

bbday sue<CR>

Table 7-14 summarizes the syntax and capabilities of the **bbday** shell program.

Table 7-14. Summary of the **bbday** Command

Shell Program Recap **bbday** – mail a banner birthday greeting (user defined)	
Command	**Arguments**
bbday	*login*
Description:	**bbday** mails the message happy birthday, in poster-sized letters, to the specified login.

The **who** command lists all users currently logged in on the system. How can you make a simple shell program called **whoson**, that will tell you if the owner of a particular login is currently working on the system?

Type the following command line into a file called **whoson**:

who | grep $1<CR>

The **who** command lists all current system users, and **grep** searches the output of the **who** command for a line containing the string contained as a value in the positional parameter **$1**.

Now try using your login as the argument for the new program **whoson**. For example, say your login is **sue**. When you issue the **whoson** command, the shell program substitutes **sue** for the parameter **$1** in your program and executes as if it were:

who | grep sue <CR>

The output is shown on the following screen:

```
$ whoson sue<CR>
sue    tty26       Jan 24 13:35
$
```

If the owner of the specified login is not currently working on the system, **grep** fails and the **whoson** prints no output.

Table 7-15 summarizes the syntax and capabilities of the **whoson** command.

Table 7-15. Summary of the **whoson** Command

Shell Program Recap	
whoson – display login information if user is logged in (user defined)	
Command	**Arguments**
whoson	*login*
Description:	If a user is on the system, **whoson** displays the user's login, the TTY number, and the time and date the user logged in.

The shell allows a command line to contain 128 arguments. However, a shell program is restricted to referencing nine positional parameters, **$1** through **$9**, at a given time. (This restriction can be worked around using the **shift** command.) The special parameter **$***, described in the next section, can also be used to access the values of all command line arguments.

Special Parameters

The **$#** is a special parameter. When referenced within a shell program, contains the number of arguments with which the shell program was invoked. Its value can be used anywhere within the shell program.

Enter the command line shown in the following screen in an executable shell program called **get.num**. Then run the **cat** command on the file:

```
$ cat get.num<CR>
echo The number of arguments is: $#
$
```

The program displays the number of arguments with which it is invoked. For example:

```
$ get.num test out this program<CR>
The number of arguments is: 4
$
```

Table 7-16 summarizes the **get.num** shell program.

Table 7-16. Summary of the **get.num** Shell Program

Shell Program Recap	
get.num – count and display the number of arguments (user defined)	
Command	**Arguments**
get.num	*(character_string)*
Description:	**get.num** counts the number of arguments given to the command and then displays the total.
Remarks:	This command demonstrates the special parameter **$#**.

The **$*** is a special parameter. When referenced within a shell program, contains a string with all the arguments with which the shell program was invoked, starting with the first. You are not restricted to nine parameters as with the positional parameters **$1** through **$9**.

You can write a simple shell program to demonstrate **$*.** Create a shell program called **show.param** that will **echo** all parameters. Use the command line shown in the following completed file:

```
$ cat show.param<CR>
echo The parameters for this command are: $*
$
```

show.param echoes all the arguments you give to the command. Make **show.param** executable and try it out, using these parameters:

Hello. How are you?

```
$ show.param Hello. How are you?<CR>
The parameters for this command are: Hello.  How are you?
$
```

Notice that **show.param** echoes `Hello. How are you?` Now try **show.param** using more than nine arguments:

```
$ show.param one two 3 4 5 six 7 8 9 10 11<CR>
The parameters for this command are: one two 3 4 5 six 7 8 9 10 11
$
```

Once again, **show.param** echoes all the arguments you give. The **$*** parameter can be useful if you use file name expansion to specify arguments to the shell command.

Use the file name expansion feature with your **show.param** command. For example, you have several files in your directory named for chapters of a book: **chap1**, **chap2**, and so on, through **chap7**. **show.param** will print a list of all those files.

```
$ show.param chap?<CR>
The parameters for this command are: chap1 chap2 chap3
chap4 chap5 chap6 chap7
$
```

Table 7-17 summarizes the **show.param** shell program.

Table 7-17. Summary of the **show.param** Shell Program

Shell Program Recap **show.param** – display all positional parameters (user defined)	
Command	**Arguments**
show.param	(any positional parameters)
Description:	**show.param** displays all the parameters.
Remarks:	If the parameters are file name generations, the command will display each of those file names.

Named Variables

Another form of variable you can use within a shell program is a *named variable.* You assign values to named variables yourself. The format for assigning a value to a named variable is:

named_variable=*value*<**CR**>

Notice that there are no spaces on either side of the = sign.

In the following example, **var1** is a named variable, and **myname** is the value or character string assigned to that variable:

var1=**myname**<**CR**>

A **$** is used in front of a variable name in a shell program to reference the value of that variable. Using the example above, the reference **$var1** tells the shell to substitute the value **myname** (assigned to **var1**) for any occurrence of the character string **$var1**.

The first character of a variable name must be a letter or an underscore. The rest of the name can be composed of letters, underscores, and digits. As in shell program file names, it is not advisable to use a shell command name as a variable name. Also, the shell has reserved some variable names you should not use for your variables. A brief explanation of these *reserved* shell variable names follows:

CDPATH
defines the search path for the **cd** command.

CERASE
is your preferred reassignment for the erase character. It is currently used only by **/local/bin/TermFuncs**.

CINTR
is your preferred reassignment for the interrupt (terminate) character.

CKILL
is your preferred reassignment for the kill character.

CQUIT
is your preferred reassignment for the quit (abort) character.

HOME
is the default variable for the **cd** command (home directory).

IFS
defines the internal field separators (normally the space, tab, and carriage return).

LOGNAME
is your login name.

MAIL
names the file that contains your electronic mail.

PATH
determines the search path used by the shell to find commands.

PS1
defines the primary prompt (default is **$**).

PS2
defines the secondary prompt (default is >).

TERM
identifies your terminal type. It is important to set this variable when you are editing with **vi** or **sledit**.

TERMINFO
identifies the directory to be searched for information about your terminal, for example, its screen size.

TZ
defines the time zone (default is **EST5EDT**).

Many of these variables are explained in *Modifying Your Login Environment* later in this chapter. You can also read more about them on the **sh**(1) manual page in the *User's Reference Manual*.

You can see the value of these variables in your shell in two ways. First, you can type:

echo $*variable_name*

The system outputs the value of *variable_name*. Second, you can use the **env**(1) command to print out the value of all defined variables in the shell. To do this, type **env** on a line by itself; the system outputs a list of the variable names and values. Typing **set** displays the value whether or not it is exported.

Assigning a Value to a Variable

If you edit with **vi**, you know you can set the **TERM** variable by entering the following command line:

TERM=*terminal_name*<**CR**>

This is the simplest way to assign a value to a variable.

There are several other ways to do this:

- Use the **read** command to assign input to the variable.
- Redirect the output of a command into a variable by using command substitution with grave accents (` ... `).
- Assign a positional parameter to the variable.

The following sections discuss each of these methods in detail.

Using the read Command

The **read** command when used within a shell program, allows you to prompt the user of the program for the values of variables. The general format for the **read** command is:

read *variable*<**CR**>

The values assigned by **read** to *variable* are substituted for **$***variable* wherever it is used in the program. If a program executes the **echo** command just before the **read** command, the program can display directions, e.g., such as `Type in ...`. The **read** command waits until you type a character string, followed by a RETURN key, then makes that string the value of the variable.

The following example shows how to write a simple shell program called **num.please** to keep track of your telephone numbers. This program uses the following commands for the purposes specified:

echo	to prompt you for a persons last name
read	to assign the input value to the variable **name**
grep	to search the file **list** for this variable

Your finished program should look like the following:

```
$ cat num.please<CR>
echo Type in the last name:
read name
grep $name list
$
```

Create a file called **list** that contains several last names and phone numbers. Then try running **num.please**.

The next example is a program called **mknum**, which creates a list. **mknum** includes the following commands for the purposes shown:

- **echo** prompts for a persons name.
- **read** assigns the persons name to the variable *name*.
- **echo** asks for the persons number.
- **read** assigns the telephone number to the variable *num*.
- **echo** adds the values of the variables *name* and *num* to the file **list**.

If you want the output of the **echo** command to be added to the end of **list**, you must use >> to redirect it. If you use >, **list** will contain only the last phone number you added.

Running the **cat** command on **mknum** displays the program's contents. When your program looks like this, you are ready to make it executable (with the **chmod** command):

```
$ cat mknum<CR>
echo Type in name
read name
echo Type in number
read num
echo $name $num >> list
$ chmod u+x mknum<CR>
$
```

Try out the new programs for your phone list. In the next example, **mknum** creates a new listing for Mr. Niceguy. Then **num.please** gives you Mr. Niceguy's phone number:

```
$ mknum<CR>
Type in the name
Mr. Niceguy<CR>
Type in the number
668-0007<CR>
$ num.please<CR>
Type in last name
Niceguy<CR>
Mr. Niceguy 668-0007
$
```

Notice that the variable **name** accepts both **Mr.** and **Niceguy** as the value.

Tables 7-18 and 7-19 summarize the **mknum** and **num.please** shell programs, respectively.

Table 7-18. Summary of the **mknum** Shell Program

Shell Program Recap **mknum** – place name and number on a phone list	
Command	**Arguments**
mknum	(interactive)
Description:	Asks you for the name and number of a person and adds that name and number to your phone list.
Remarks:	This is an interactive command.

Table 7-19. Summary of the **num.please** Shell Program

Shell Program Recap **num.please** – display a person's name and number	
Command	**Arguments**
num.please	(interactive)
Description:	Asks you for a persons last name, and then displays the persons full name and telephone number.
Remarks:	This is an interactive command.

Substituting Command Output for the Value of a Variable

You can substitute a commands output for the value of a variable by using *command substitution.* Use the following format:

variable=`*command*`<**CR**>

The output from *command* becomes the value of *variable.*

In one of the previous examples on piping, the **date** command was piped into the **cut** command to get the correct time. That command line was:

date | cut -c12-19<**CR**>

You can put this in a simple shell program called **t** that will give you the time.

```
$ cat t<CR>
time=`date | cut -c12-19`
echo The time is: $time
$
```

Remember there are no spaces on either side of the equal sign. Make the file executable, and you will have a program that gives you the time:

```
$ chmod u+x t<CR>
$ t<CR>
The time is: 10:36
$
```

Table 7-20 summarizes your **t** program.

Table 7-20. Summary of the **t** Shell Program

Shell Program Recap **t** – display the correct time	
Command	**Arguments**
t	none
Description:	**t** gives you the correct time in hours and minutes.

Assigning Values with Positional Parameters

You can assign a positional parameter to a named parameter by using the format:

var1 = **$1<CR>**

The next example is a simple program called **simp.p** that assigns a positional parameter to a variable:

```
$ cat simp.p<CR>
var1=$1
echo $var1
$
```

You can also assign the output of a command that uses positional parameters to a variable:

person=`who | grep $1`<CR>

In the next example, the program **log.time** keeps track of your **whoson** program results. The output of **whoson** is assigned to the variable **person**, and added to the file **login.file** with the **echo** command. The last **echo** displays the value of **$person**, which is the same as the output from the **whoson** command:

```
$ cat log.time<CR>
person=`who | grep $1`
echo $person >> login.file
echo $person
$
```

The system response to **log.time** is shown in the following screen:

```
$ log.time maryann<CR>
maryann      tty61          Apr 11 10:26
$
```

Table 7-21 summarizes the **log.time** shell program.

Table 7-21. Summary of the **log.time** Shell Program

Shell Program Recap **log.time** – log and display a specified login (user defined)	
Command	**Arguments**
log.time	*login*
Description:	If the specified login is currently on the system, **log.time** places the line of information from the **who** command into the file **login.file** and then displays that line of information on your terminal.

Shell Programming Constructs

The shell programming language has several constructs that give added flexibility to your programs:

- Comments let you document a programs function.
- The **here** document allows you to include within the shell program itself lines to be redirected to be the input to some command in the shell program.

- The **exit** command lets you terminate a program at a point other than the end of the program and use return codes.
- The looping constructs, **for** and **while**, allow a program to iterate through groups of commands in a loop.
- The conditional control commands, **if** and **case**, execute a group of commands only if a particular set of conditions is met.
- The **break** command allows a program to exit unconditionally from a loop.

Comments

You can place comments in a shell program in two ways. All text on a line following a # (pound) sign is ignored by the shell. The # sign can be at the beginning of a line, in which case the comment uses the entire line, or it can occur after a command, in which case the command is executed but the remainder of the line is ignored. The end of a line always ends a comment. The general format for a comment line is:

#comment<**CR**>

For example, a program that contains the following lines ignores them when it is executed:

```
#This program sends a generic birthday greeting.<CR>
#This program needs a login as<CR>
#the positional parameter.<CR>
```

Comments are useful for documenting a programs function and should be included in any program you write.

NOTE

If the # sign is your erase character **(CERASE)**, you must precede it with a \ when creating program comments.

The here Document

A **here** document allows you to place into a shell program lines that are redirected to be the input of a command in that program. It is a way to provide input to a command in a shell program without needing to use a separate file. The notation consists of the redirection symbol << and a delimiter that specifies the beginning and end of the lines of input. The delimiter can be one character or a string of characters; the **!** is often used.

Figure 7-1 shows the general format for a **here** document.

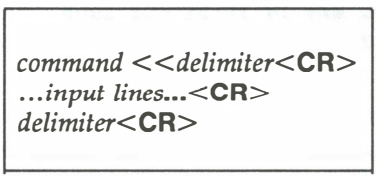


Figure 7-1. Format of a **here** Document

In the next example, the program **gbday** uses a **here** document to send a generic birthday greeting by redirecting lines of input into the **mail** command:

```
$ cat gbday<CR>
mail $1 <<!
Best wishes to you on your birthday.
!
$
```

When you use this command, you must specify the recipients login as the argument to the command. The input included with the use of the **here** document is:

```
Best wishes to you on your birthday
```

For example, to send this greeting to the owner of login **mary**, type:

```
$ gbday mary<CR>
```

Login **mary** will receive your greeting the next time she reads her mail messages:

```
$ mail<CR>
From mylogin Wed May 14 14:31 CDT 1986
Best wishes to you on your birthday
$
```

Table 7-22 summarizes the format and capabilities of the **gbday** command.

Table 7-22. Summary of the **gbday** Command

Shell Program Recap **gbday -** send a generic birthday greeting (user defined)	
Command	**Arguments**
gbday	*login*
Description:	**gbday** sends a generic birthday greeting to the owner of the login specified in the argument.

Using ed in a Shell Program

The **here** document offers a convenient and useful way to use **ed** in a shell script. For example, suppose you want to make a shell program that will enter the **ed** editor, make a global substitution to a file, write the file, and then quit **ed**. The following screen shows the contents of a program called **ch.text** that does these tasks:

```
$ cat ch.text<CR>
echo Type in the file name.
read file1
echo Type in the exact text to be changed.
read old_text
echo Type in the exact new text to replace the above.
read new_text
ed - $file1 <<!
g/$old_text/s//$new_text/g
w
q
!
$
```

Notice the – (minus) option to the **ed** command. This option prevents the character count from being displayed on the screen. Notice, also, the format of the **ed** command for global substitution:

g/*old_text*/s//*new_text*/**g<CR>**

The program uses three variables: *file1*, *old_text*, and *new_text*. When the program is run, it uses the **read** command to obtain the values of these variables. The variables provide the following information:

file	the name of the file to be edited
old_text	the exact text to be changed
new_text	the new text

Once the variables are entered in the program, the **here** document redirects the global substitution, the **write** command, and the **quit** command into the **ed** command. Try the new **ch.text** command. The following screen shows sample responses to the program prompts:

```
$ ch.text<CR>
Type in the filename.
memo<CR>
Type in the exact text to be changed.
Dear John:<CR>
Type in the exact new text to replace the above.
To whom it may concern:<CR>
$ cat memo<CR>
To whom it may concern:
$
```

Notice that by running the **cat** command on the changed file, you could examine the results of the global substitution.

Table 7-23 summarizes the format and capabilities of the **ch.text** command.

Table 7-23. Summary of the **ch.text** Command

Shell Program Recap **ch.text** – change text in a file	
Command	**Arguments**
ch.text	(interactive)
Description:	Replaces text in a file with new text.
Remarks:	This shell program is interactive. It prompts you to type in the arguments.

If you want to become more familiar with **ed**, see Chapter 5, *Line Editor Tutorial (ed).* The stream editor **sed** can also be used in shell programming.

Return Codes

Most shell commands issue return codes that indicate whether the command executed properly. By convention, if the value returned is 0 (zero), the command executed properly; any other value indicates that it did not. The return code is not printed automatically, but is available as the value of the shell special parameter **$?**.

Checking Return Codes

After executing a command interactively, you can see its return code by typing

```
echo $?
```

Consider the following example:

```
$ cat hi
This is file hi.
$ echo $?
0
$ cat hello
cat: cannot open hello
$ echo $?
2
$
```

In the first case, the file **hi** exists in your directory and has read permission for you. The **cat** command behaves as expected and outputs the contents of the file. It exits with a return code of 0, which you can see using the parameter **$?**. In the second case, the file either does not exist or does not have read permission for you. The **cat** command prints a diagnostic message and exits with a return code of 2.

Using Return Codes with the exit Command

A shell program normally terminates when the last command in the file is executed. However, you can use the **exit** command to terminate a program at some other point. Perhaps more importantly, you can also use the **exit** command to issue return codes for a shell program. (See the **exit**(2) manual page in the *Programmer's Reference Manual*.)

Looping

In the previous examples in this chapter, the commands in shell programs have been executed in sequence. The **for** and **while** looping constructs allow a program to execute a command or sequence of commands several times.

The for Loop

The **for** loop executes a sequence of commands once for each member of a list. It has the following format:

```
for variable<CR>
      in a_list_of_values<CR>
do<CR>
      command 1<CR>
      command 2<CR>
            .
            .
            .
      last command<CR>
done<CR>
```

Figure 7-2. Format of the **for** Loop Construct

For each iteration of the loop, the next member of the list is assigned to the variable given in the **for** clause. References to that variable may be made anywhere in the commands within the **do** clause.

It is easier to read a shell program if the looping constructs are visually clear. Since the shell ignores spaces at the beginning of lines, each section of commands can be indented as it was in the above format. Also, if you indent each command section, you can easily check to make sure each **do** has a corresponding **done** at the end of the loop.

The variable can be any name you choose. For example, if you call it **var**, then the values given in the list after the keyword **in** will be assigned in turn to **var**; references within the command list to **$var** will make the value available. If the **in** clause is omitted, the values for **var** will be the complete set of arguments given to the command and available in the special parameter **$***. The command list between the keywords **do** and **done** will be executed once for each value.

When the commands have been executed for the last value in the list, the program executes the next line below **done**. If there is no line, the program ends.

The easiest way to understand a shell programming construct is to try an example. Create a program that moves files to another directory. To do this, include the following commands.

echo
to prompt the user for a path name to the new directory.

read
to assign the path name to the variable **path**.

for *variable*
to call the variable **file**; it can be referenced as **$file** in the command sequence.

in *list_of_values*
to supply a list of values. If the **in** clause is omitted, the list of values is assumed to be **$*** (all arguments entered on the command line).

do *command_sequence*
to provide a command sequence. The construct for this program is:

```
do
    mv $file $path/$file<CR>
done
```

The following screen shows the text for the shell program **mv.file**:

```
$ cat mv.file<CR>
echo Please type in the directory path
read path
for file
   in memo1 memo2 memo3
do
   mv $file $path/$file
done
$
```

In this program the values for the variable **file** are already in the program. To change the files each time the program is invoked, assign the values using positional parameters or the **read** command. When positional parameters are used, the **in** keyword is not needed, as the next screen shows:

```
$ cat mv.file<CR>
echo type in the directory path
read path
for file
do
   mv $file $path/$file
done
$
```

You can move several files at once with this command by specifying a list of file names as arguments to the command. (This can be done most easily using the file name expansion mechanism described earlier).

Table 7-24 summarizes the **mv.file** shell program.

Table 7-24. Summary of **mv.file** Shell Program

Shell Program Recap **mv.file** – move files to another directory (user defined)	
Command	**Arguments**
mv.file	*filenames* (interactive)
Description:	Moves files to a new directory.
Remarks:	This program requires file names to be given as arguments. The program prompts for the path to the new directory.

The while Loop

Another loop construct, the **while** loop, uses two groups of commands. It will continue executing the sequence of commands in the second group, the **do...done** list, as long as the final command in the first group, the **while** list, returns a status of true (meaning the command can be executed).

The general format of the **while** loop is shown in Figure 7-3.

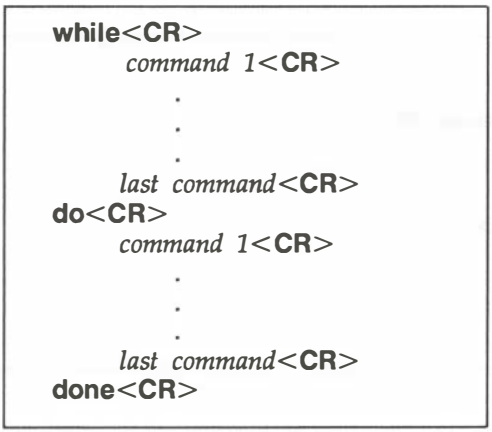


Figure 7-3. Format of the **while** Loop Construct

NOTE

The <**CR**> following **while** is optional. White space will do, but multiple commands must be separated by ; or <**CR**>.

For example, a program called **enter.name** uses a **while** loop to enter a list of names into a file. The program consists of the following command lines:

```
$ cat enter.name<CR>
while
   read x
do
   echo $x>>xfile
done
$
```

With some added refinements, the program becomes:

```
$ cat enter.name<CR>
echo Please type in each person's name and then a <CR>
echo Please end the list of names with a <^d>
while read x
do
   echo $x>>xfile
done
echo xfile contains the following names:
cat xfile
$
```

Notice that after the loop is completed, the program executes the commands below the **done**.

You used special characters in the first two **echo** command lines, so you must use quotes to turn off the special meaning. The next screen shows the results of **enter.name**:

```
$ enter.name<CR>
Please type in each person's name and then a <CR>
Please end the list of names with a <^d>
Mary Lou<CR>
Janice<CR>
<^d>
xfile contains the following names:
Mary Lou
Janice
$
```

Notice that after the loop completes, the program prints all the names contained in **xfile**.

The Shell's Garbage Can: /dev/null

The file system has a file called **/dev/null** where you can have the shell deposit any unwanted output. Try out **/dev/null** by destroying the results of the **who** command. First, type in the **who** command. The response tells you who is on the system. Now, try the **who** command, but redirect the output into **/dev/null**:

who > /dev/null<CR>

Notice that the system responded with a prompt. The output from the **who** command was placed in **/dev/null** and was effectively discarded.

if...then

The **if** command tells the shell program to execute the **then** sequence of commands only if the final command in the **if** command list is successful. The **if** construct ends with the keyword **fi**.

The general format for the **if** construct is shown in Figure 7-4.

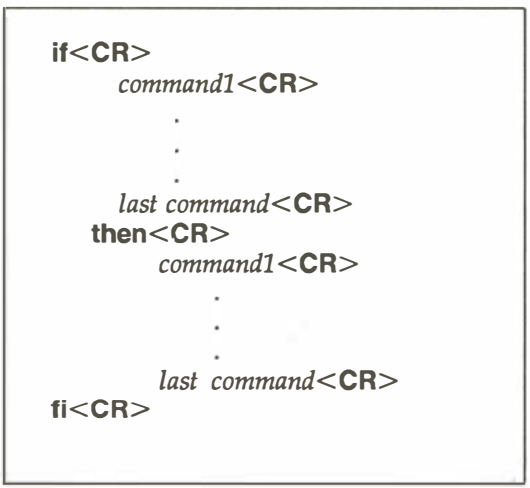


Figure 7-4. Format of the **if...then** Conditional Construct

NOTE

The <**CR**> following **if** is optional. Any white space will do, but multiple commands must be separated by ; or <**CR**>.

For example, a shell program called **search** demonstrates the use of the **if...then** construct. **search** uses the **grep** command to search for a word in a file. If **grep** is successful, the program will **echo** that the word is found in the file. Copy the **search** program (shown on the following screen) and try it yourself:

```
$ cat search<CR>
echo Type in the word and the file name.
read word file
if grep $word $file
   then echo $word is in $file
fi
$
```

Notice that the **read** command assigns values to two variables. The first characters you type, up until a space, are assigned to **word**. The rest of the characters, including embedded spaces, are assigned to **file**.

A problem with this program is the unwanted display of output from the **grep** command. If you want to dispose of the system response to the **grep** command in your program, use the file **/dev/null**, changing the **if** command line to the following:

if grep $*word* **$***file* **> /dev/null<CR>**

Now execute your **search** program. It should respond only with the message specified after the **echo** command.

if...then...else

The **if...then** construction can also issue an alternate set of commands with **else**, when the **if** command sequence is false. It has the following general format:

```
if<CR>
      command1<CR>
            .
            .
            .
      last command<CR>
   then<CR>
      command1<CR>
            .
            .
            .
      last command<CR>
   else<CR>
      command1<CR>
            .
            .
            .
      last command<CR>
fi<CR>
```

Figure 7-5. Format of the **if...then...else** Conditional Construct

You can now improve your **search** command so it will tell you when it cannot find a word, as well as when it can. The following screen shows how your improved program will look:

```
$ cat search<CR>
echo Type in the word and the file name.
read word file
if
   grep $word $file >/dev/null
then
   echo $word is in $file
else
   echo $word is NOT in $file
fi
$
```

Table 7-25 summarizes your enhanced **search** program.

Table 7-25. Summary of the **search** Shell Program

Shell Program Recap **search** - tells you if a word is in a file (user defined)	
Command	**Arguments**
search	interactive
Description:	Reports whether a word is in a file.
Remarks:	The command prompts you for the arguments (the word and the file).

The test Command for Loops

The **test** command, which checks to see if certain conditions are true, is a useful command for conditional constructs. If the condition is true, the loop continues. If the condition is false, the loop ends and the next command is executed. Some of the useful options for the **test** command are:

test **−r** *file*<**CR**>
true if the file exists and is readable

test **−w** *file*<**CR**>
true if the file exists and has write permission

test **−x** *file*<**CR**>
true if the file exists and is executable

test **−s** *file*<**CR**>
true if the file exists and has at least one character

test *var1* **−eq** *var2*<**CR**>
true if *var1* equals *var2*

test *var1* **−ne** *var2*<**CR**>
true if *var1* does not equal *var2*

You may want to create a shell program to move all the executable files in the current directory to your **bin** directory. You can use the **test -x** command to select the executable files. Review the example of the **for** construct that occurs in the **mv.file** program, shown in the following screen:

```
$ cat mv.file<CR>
echo type in the directory path
read path
for file
do
  mv $file $path/$file
done
$
```

Create a program called **mv.ex** that includes an **if test –x** statement in the **do...done** loop to move executable files only:

```
$ cat mv.ex<CR>
echo type in the directory path
read path
for file
  do
    if test -x $file
       then
         mv $file $path/$file
    fi
  done
$
```

The directory path will be the path from the current directory to the **bin** directory. However, if you use the value for the shell variable **HOME**, you do not need to type in the path each time. **$HOME** gives the path to the login directory. **$HOME/bin** gives the path to your **bin**.

In the following example, **mv.ex** does not prompt you to type in the directory name, and therefore, does not read the **path** variable:

```
$ cat mv.ex<CR>
for file
  do
    if test -x $file
       then
         mv $file $HOME/bin/$file
    fi
  done
$
```

Test the command, using all files in the current directory, specified with the * metacharacter as the command argument. The command lines shown in the following example execute the command from the current directory, then changes to **bin** and lists the files in that directory. All executable files should be there.

```
$ mv.ex *<CR>
$ cd; cd bin; ls<CR>
list_of_executable_files
$
```

Table 7-26 summarizes the format and capabilities of the **mv.ex** shell program.

Table 7-26. Summary of the **mv.ex** Shell Program

Shell Program Recap **mv.ex** – move all executable files in the current directory to the **bin** directory	
Command	**Arguments**
mv.ex	* (all file names)
Description:	Moves all files in the current directory with execute permission to the **bin** directory.
Remarks:	All executable files in the **bin** directory (or any directory shown by the **PATH** variable) can be executed from any directory.

case..esac

The **case...esac** construction has a multiple-choice format that allows you to choose one of several patterns, then execute a list of commands for that pattern. The pattern statements must begin with the keyword **in**, and a right parenthesis, **)**, must be placed after the last character of each pattern. The command sequence for each pattern is ended with two semicolons, **;;**. The **case** construction must be ended with **esac** (the letters of the word case reversed).

The general format for the **case** construction is shown in Figure 7-6.

```
case word<CR>
in<CR>
    pattern1)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    pattern2)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    pattern3)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    *)<CR>
        command 1<CR>
            .
            .
            .
        last command<CR>
    ;;<CR>
esac<CR>
```

Figure 7-6. The **case...esac** Conditional Construct

The **case** construction tries to match the *word* following the word **case** with the *pattern* in the first pattern section. If there is a match, the program executes the command lines after the first pattern and up to the corresponding **;;**.

If the first pattern is not matched, the program proceeds to the second pattern. Once a pattern is matched, the program does not try to match any more of the patterns, but goes to the command following **esac**.

The * used as a pattern matches any *word*, and allows you to give a set of commands to be executed if no other pattern matches. To do this, it must be placed as the last possible pattern in the **case** construct, so that the other patterns are checked first. This provides a useful way to detect erroneous or unexpected input.

The patterns that can be specified in the *pattern* part of each section may use the metacharacters *, **?**, and **[ ]** as described earlier in this chapter for the shells file name expansion capability. This provides useful flexibility.

The **set.term** program contains a good example of the **case...esac** construction. This program sets the shell variable TERM according to the type of terminal you are using. It uses the following command line:

TERM=*terminal_name*<**CR**>

(For an explanation of the commands used, see the **vi** tutorial in Chapter 6.) In the following example, the terminal is a Teletype 4420, Teletype 5410, or Teletype 5420.

set.term first checks to see whether the value of **term** is 4420. If it is, the program makes T4 the value of **TERM** and terminates. If it the value of **term** is not 4420, the program checks for other values: 5410 and 5420. It executes the commands under the first pattern that it finds, and then goes to the first command after the **esac** command.

The pattern *, meaning everything else, is included at the end of the terminal patterns. It warns that you do not have a pattern for the terminal specified and allows you to exit the **case** construct.

```
$ cat set.term<CR>
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
       in
           4420)
               TERM=T4
           ;;
           5410)
               TERM=T5
           ;;
           5420)
               TERM=T7
           ;;
           *)
           echo not a correct terminal type
           ;;
esac
export TERM
echo end of program
$
```

Notice the use of the **export** command. You use **export** to make a variable available within your environment and to other shell procedures. What would happen if you placed the * pattern first? The **set.term** program would never assign a value to **TERM**, since it would always match the first pattern *, which means everything.

Table 7-27 summarizes the format and capabilities of the **set.term** shell program.

Table 7-27. Summary of the **set.term** Shell Program

Shell Program Recap **set.term** - assign a value to TERM (user defined)	
Command	**Arguments**
set.term	interactive
Description:	Assigns a value to the shell variable TERM, then exports that value to other shell procedures.
Remarks:	This command asks for a specific terminal code to be used as a pattern for the **case** construction.

NOTE

A program to set and export a variable is useful only within programs that call other shell programs (or binaries) that use it. After completion, the value of **TERM** is lost unless it is executed as follows:

$ .set.term<CR>

Unconditional Control Statements: break and continue Commands

The **break** command unconditionally stops the execution of any loop in which it is encountered and goes to the next command after the **done, fi,** or **esac** statement. If there are no commands after that statement, the program ends.

In the example for **set.term**, you could have used the **break** command instead of **echo** to leave the program, as shown in the next example.

```
$ cat set.term<CR>
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
        in
            4420)
            TERM=T4
        ;;
        5410)
            TERM=T5
        ;;
        5420)
            TERM=T7
        ;;
        *)
            break
        ;;
esac
export TERM
echo end of program
$
```

The **continue** command causes the program to go immediately to the next iteration of a **do** or **for** loop without executing the remaining commands in the loop.

Debugging Programs

At times you may need to debug a program to find and correct errors. There are two options to the **sh** command (listed below) that can help you debug a program:

sh −v *shellprogramname*
prints the shell input lines as they are read by the system.

sh −x *shellprogramname*
prints commands and their arguments as they are executed.

To try out these two options, create a shell program that has an error in it. For example, create a file called **bug** that contains the following list of commands:

```
$ cat bug<CR>
today=`date`
echo enter person
read person
mail $1
$person
When you log off come into my office please.
$today.
MLH
$
```

Notice that **today** equals the output of the **date** command, which must be enclosed in grave accents for command substitution to occur.

The mail message sent to Tom **($1)** at login **tommy ($2)** should read as the following screen shows:

```
$ mail<CR>
From mlh  Thu  Apr 10  11:36  CST  1984
Tom
When you log off come into my office please.
Thu  Apr 10  11:36:32  CST  1986
MLH
?
```

If you try to execute **bug**, you will have to press the **BREAK** or **CINTR** key to end the program.

To debug this program, try executing **bug** using **sh –v**. This prints the lines of the file as they are read by the system:

```
$ sh -v bug tom<CR>
today=`date`
echo enter person
enter person
read person
tommy
mail $1
```

Notice that the output stops on the **mail** command, since there is a problem with **mail**. You must use the **here** document to redirect input into **mail**.

Before you fix the **bug** program, try executing it with **sh –x**, which prints the commands and their arguments as they are read by the system:

```
$ sh -x bug tom tommy<CR>
+date
today=Thu Apr 10 11:07:23 CST 1986
+ echo enter person
enter person
+ read person
tommy
+ mail tom
$
```

Once again, the program stops at the **mail** command. Notice that the substitutions for the variables have been made and are displayed.

The corrected **bug** program is:

```
$ cat bug<CR>
today=`date`
echo enter person
read person
mail $1 <<!
$person
When you log off come into my office please.
$today
MLH
!
$
```

The **tee** command is a helpful command for debugging pipelines. While simply passing its standard input to its standard output, it also saves a copy of its input into the file whose name is given as an argument.

The general format of the **tee** command is:

command1 | **tee** *saverfile* | *command2*<**CR**

The output of *command* is saved in the file *saverfile.*

For example, you want to check on the output of the **grep** command in the following command line:

who | grep $1 | cut -c1-9<CR>

You can use **tee** to copy the output of **grep** into a file called **check**, without disturbing the rest of the pipeline.

who | grep $1 | tee check | cut -c1-9<CR>

The file **check** contains a copy of the **grep** output:

```
$ who | grep mlhmo | tee check | cut -c1-9<CR>
mlhmo
$ cat check<CR>
mlhmo    tty61    Apr 10    11:30
$
```

Modifying Your Login Environment

The SYSTEM V/88 operating system lets you modify your login environment in several ways. One modification that users commonly want is to change the default values of the erase (#) and line kill (@) characters.

Normal or **gtty** (by-products of **TermFuncs**) may also be invoked at any later time to reset or view these values.

When you log in, the shell first examines a file in your login directory named **.profile**. This file contains commands that control your shell environment.

Because the **.profile** is a file, it can be edited and changed to suit your needs. On some systems you can edit this file yourself, while on others, the System Administrator does this for you. To see whether you have a **.profile** in your home directory, type:

ls –al $HOME

If you can edit the file yourself, you may want to be cautious the first few times. Before making any changes to your **.profile**, make a copy of it in another file called **safe.profile**. Type:

cp .profile safe.profile<CR>

You can add commands to your **.profile** just as you add commands to any other shell program. You can also set some terminal options with the **stty** command, and set some shell variables.

Adding Commands to Your .profile

Practice adding commands to your **.profile**. Edit the file and add the following **echo** command to the last line of the file:

echo Good Morning! I am ready to work for you.

Write and quit the editor.

Whenever you make changes to your **.profile** and you want to initiate them in the current work session, you may cause the commands in **.profile** to be executed directly using the **.** (dot) shell command. The shell reinitializes your environment by reading executing the commands in your **.profile**. Try this now, type:

. .profile<CR>

The system should respond with the following:

```
Good Morning! I am ready to work for you.
$
```

Reassigning the Delete Functions

You can reassign the erase and kill characters in two ways: directly using **stty(1)**, or through the Terminal Support System. Using the Terminal Support System is preferred because it generates a shell function to make it easier to reset the environment to your desired state. The Terminal Support System is described in Appendix F.

Your **.profile** must be modeled after, or be copied from, **/etc/stdprofile** (the normal case) for this method to work. Specifically, you must set the values of four shell variables to the desired line-editing characters and invoke the **TermFuncs** command and the **normal** function. For example:

```
export CERASE CKILL CINTR CQUIT
CERASE="^H"
CKILL="^U"
CINTR=          #Retains the default
CQUIT=          #Retains the default
. TermFuncs     #Define function 'normal'
normal          #Invoke it

***
erase = ^h kill = ^u intr = ^? quit = ^|
***
```

Normal or **gtty** (by-products of **TermFuncs**) may also be invoked at any later time to reset or view these values.

The erase, kill, intr, and quit characters may be modified with **stty**(1) (this is only an example):

```
stty erase "^H"
stty kill "^U"
stty intr "^X"
stty quit "^C"
```

These may be combined in one **stty** using the following:

```
stty erase "^H" kill "^U" intr "^X" quit "^C"
```

These statements may be included in your **.profile**.

Setting Terminal Options

The **stty** command can make your shell environment more convenient. By using the **stty −tabs** option, you can preserve tabs when printing. This option expands the tab setting to eight spaces, which is the default. The number of spaces for each tab can be changed. (See **stty**(1) in the *User's Reference Manual* for details.)

If you want to use this option for the **stty** command, you can create the command lines in your **.profile** just as you would create them in a shell program. If you use the **tail** command, which displays the last few lines of a file, you can see the results of adding this command line to your **.profile**:

```
$ tail -4 .profile<CR>
echo Good Morning! I am ready to work for you
stty -tabs
$
```

Table 7-28 summarizes the format and capabilities of the **tail** command.

Table 7-28. Summary of the **tail** Command

Command Recap **tail** – display the last portion of a file		
Command	**Options**	**Arguments**
tail	*–n*	*filename*
Description:	Displays the last lines of a file.	
Options:	Use *–n* to specify the number of lines *n* (default is 10 lines). You can specify a number of blocks (**–***n***b**) or characters (**–***n***c**) instead of lines.	

Creating a Public Directory

We have often talked about sharing useful programs with other users in this chapter. Similarly, these users may have programs or other files that they want to share with you. So that these users can send you the files easily, you should create a **public** directory:

```
mkdir public
chmod go+w public
```

Notice that you have to change the permissions of the directory using **chmod**. When you have a **public** directory with the correct permissions, other users can send you files using the **uucp** command. See the **uucp**(1) manual page in the *User's Reference Manual* for details.

Using Shell Variables

Several of the variables reserved by the shell are used in your **.profile**. You can display the current value for any shell variable by entering the following command:

echo $*variable_name*

Four of the most basic of these variables are discussed next.

HOME

This variable gives the path name of your login directory. Use the **cd** command to go to your login directory and type:

pwd<CR>

What was the system response? Now type:

echo $HOME<CR>

Was the system response the same as the response to **pwd**?

$HOME is the default argument for the **cd** command. If you do not specify a directory, **cd** will move you to **$HOME**.

PATH

This variable gives the search path for finding and executing commands. To see the current values for your **PATH** variable, type:

echo $PATH<CR>

The system responds with your current PATH value.

```
$ echo $PATH<CR>
:/mylogin/bin:/bin:/usr/bin:/usr/lib
$
```

The colon (**:**) is a delimiter between path names in the string assigned to the **$PATH** variable. When nothing is specified before a **:** , the current directory is understood. Notice how, in the last example, the system looks for commands in the current directory first, then in **/mylogin/bin/**, then in **/bin**, then in **/usr/bin**, and finally in **/usr/lib**.

If you are working on a project with several other people, you may want to set up a group **bin**, a directory of special shell programs used only by your project members. The path might be named **/project1/bin**. Edit your **.profile**, and add **:/project1/bin** to the end of your **PATH**, as in the next example.

PATH=":/mylogin/bin:/bin:/usr/lib:/project1/bin"<CR>

The default **PATH** as set by **login**(1) is:

:/bin:/usr/bin # for normal users
:/bin:/usr/bin:/etc # for root

When invoked, **.profile** and **/etc/stdprofile** add **:/local/bin:/usr/local/bin** to this list.

TERM

This variable tells the shell what kind of terminal you are using. To assign a value to it, you must execute the following three commands in this order:

TERM=*terminal_name*<**CR**>
export TERM<**CR**>
TermSetup

The first two lines are necessary to tell the computer what type of terminal you are using. The last line, containing the **TermSetup** command, tells the terminal that the computer is expecting to communicate with the type of terminal specified in the **TERM** variable. Therefore, this command must always be entered after the variable has been exported.

If you do not want to specify the **TERM** variable each time you log in, add these three command lines to your **.profile**; they are executed automatically whenever you log in. To determine what terminal name to assign to the **TERM** variable, see the instructions in Appendix F, *Setting Up the Terminal*. This appendix also contains details about the **TermSetup** command and the Terminal Support programs.

If you log in on more than one type of terminal, it would also be useful to have your **set.term** command in your **.profile**.

PS1

This variable sets the primary shell prompt string (the default is the **$** sign). You can change your prompt by changing the **PS1** variable in your **.profile**.

Try the following example. Note that to use a multi-word prompt, you must enclose the phrase in quotes. Type the following variable assignment in your **.profile**.

PS1="Your command is my wish<**CR**>**"**

Now execute your **.profile** (with the **.** command) and watch for your new prompt sign.

$ . .profile<**CR**>
Your command is my wish

This is the prompt for your login shell until you delete the **PS1** variable from your **.profile**.

Shell Programming Exercises

2-1. Create a shell program called **btime** from the following command line:

banner `date | cut -c12-19`<CR>

2-2. Write a shell program that will give only the date in a banner display. Be careful not to give your program the same name as a SYSTEM V/88 system command.

2-3. Write a shell program that will send a note to several people on your system.

2-4. Redirect the **date** command without the time into a file.

2-5. Echo the phrase "Dear colleague" in the same file that contains the date command, without erasing the date.

2-6. Using the above exercises, write a shell program that will send a memo to the same people on your system mentioned in Exercise 2-3. Include in your memo:

The current date and the words "Dear colleague" at the top of the memo

The body of the memo (stored in an existing file)

The closing statement

2-7. How can you **read** variables into the **mv.file** program?

2-8. Use a **for** loop to move a list of files in the current directory to another directory. How can you move all your files to another directory?

2-9. How can you change the program **search**, so that it searches through several files?

Hint:

for file in $*

2-10. Change your prompt to the word Hello.

2-11. Check the settings of the variables **$HOME**, **$TERM**, and **$PATH** in your environment.

Answers to Exercises

Answers to Command Language Exercises

1-1. The * at the beginning of a file name refers to all files that end in that file name, including that file name.

```
$ ls *t<CR>
cat
123t
new.t
t
$
```

1-2. The command **cat [0-9]*** produces the following output:

```
1memo
100data
9
05name
```

The command **echo *** produces a list of all files in the current directory.

1-3. You can place **?** in any position in a file name.

1-4. The command **ls [0-9]*** lists only those files that start with a number.

The command **ls [a-m]*** lists only those files that start with the letters "a" through "m".

1-5. If you placed the sequential command line in the background mode, the immediate system response was the PID number for the job.

No, the **&** (ampersand) must be placed at the end of the command line.

1-6. The command line would be:

cd; pwd > junk; ls >> junk; ed trial<CR>

1-7. Change the **–c** option of the command line to read:

banner `date | cut –c1-10`<CR>

Answers to Shell Programming Exercises

2-1.

```
$ cat btime<CR>
banner `date | cut -c12-19`
$
$ chmod u+x btime<CR>
$ btime<CR>
(banner display of the time 10:26)
$
```

2-2.

```
$ cat mydate<CR>
banner `date | cut -c1-10`
$
```

2-3.

```
$ cat tofriends<CR>
echo Type in the name of the file containing the note.
read note
mail janice marylou bryan < $note
$
```

If you used parameters for the logins, instead of the logins themselves, your program may have looked like this:

```
$ cat tofriends<CR>
echo Type in the name of the file containing the note.
read note
mail $* < $note
$
```

2-4. **date | cut −c1-10 > file1<CR>**

2-5. **echo Dear colleague >> file1<CR>**

2-6.

```
$ cat send.memo<CR>
date | cut −c1-10 > memo1
echo Dear colleague >> memo1
cat memo >> memo1
echo A memo from M. L. Kelly >> memo1
mail janice marylou bryan < memo1
$
```

2-7.

```
$ cat mv.file<CR>
echo type in the directory path
read path
echo type in file names, end with <^d>
  while
  read file
    do
      mv $file $path/$file
    done
echo all done
$
```

7

2-8.

```
$ cat mv.file<CR>
echo Please type in directory path
read path
for file in $*
  do
    mv $file $path/$file
  done
$
```

The command line for moving all files in the current directory is:

$ mv.file *<CR>

2-9. See hint given with exercise 2-9.

```
$ cat search<CR>
for file
  in $*
  do
    if grep $word $file >/dev/null
    then echo $word is in $file
    else echo $word is NOT in $file
    fi
  done
$
```

2-10. Add the following command lines to your **.profile**

PS1=Hello<CR>
export PS1

2-11. To check the values of these variables in your home environment:

$ echo $HOME<CR>
$ echo $TERM<CR>
$ echo $PATH<CR>

8 Electronic Mail Tutorial

Introduction

The operating system offers a choice of commands that enable you to communicate with other operating system users. Specifically, they allow you to send and receive messages from other users (on either your system or another operating system, exchange files, and form networks with other operating systems. Through networking, a user on one system can exchange messages and files between computers, and execute commands on remote computers.

To help you take advantage of these capabilities, this chapter will teach you how to use the following commands.

For exchanging messages:	**mail**, **mailx**, **uname**, and **uuname**
For exchanging files:	**uucp**, **uuto**, **uupick**, and **uustat**
For networking:	**ct**, **cu**, and **uux**

Exchanging Messages

To send messages, you can use either the **mail** or **mailx** command. These commands deliver your message to a file belonging to the recipient. When the recipient logs in (or while already logged in), he or she receives a message that says `you have mail`. The recipient can use either the **mail** or **mailx** command to read your message and reply at their leisure.

The main difference between **mail** and **mailx** is that only **mailx** offers the following features:

- a choice of text editors (**ed** or **vi**) for handling incoming and outgoing messages
- a list of waiting messages that allows the user to decide which messages to handle and in what order
- several options for saving files
- commands for replying to messages and sending copies (of both incoming and outgoing messages) to other users

You can also use **mail** or **mailx** to send short files containing memos, reports. However, if you want to send someone a file that is over a page long, use one of the commands designed for transferring files: **uuto** or **uucp**. (See *Sending Large Files* later in this chapter for descriptions of these commands.)

The mail Facility

This section presents the **mail** command. It discusses the basics of sending mail to one or more people simultaneously, whether they are working on the local system (the same system as you) or on a remote system. It also covers receiving and handling incoming mail.

Sending Messages

The basic command line format for sending mail is:

mail *login*<**CR**>

where *login* is the recipient's login name. This login name can be either of the following:

- a login name if the recipient is on your system (e.g., **bob**)
- a system name, an exclamation point (!), and login name if the recipient is on another system that can communicate with yours (e.g., **sys2!bob**)

For the moment, assume that the recipient is on the local system. Type the **mail** command at the system prompt, press the RETURN key, and start typing the text of your message on the next line. There is no limit to the length of your message. When you have finished typing it, send the message by typing a period (.) or a <^d> (control-d) at the beginning of a new line.

The following example shows how this procedure appears on your screen:

```
$ mail phyllis<CR>
My meeting with Smith's<CR>
group tomorrow has been moved<CR>
up to 3:00 so I won't be able to<CR>
see you then. Could we meet<CR>
in the morning instead?<CR>
.<CR>
$
```

The prompt on the last line means that your message has been queued (placed in a waiting line of messages) and will be sent.

Undeliverable Mail

If you make an error when typing the recipients login, the **mail** command will not be able to deliver your mail. Instead, it prints two messages telling you that it has failed and that it is returning your mail. Then it returns your mail in a message that includes the system name and login name of both the sender and intended recipient, and an error message stating the reason for the failure.

For example, you (owner of the login **kol**) want to send a message to a user with the login **chris** on a system called **marmaduk**. Your message says **The meeting has been changed to 2:00**. Failing to notice that you have incorrectly typed the login as **cris**, you try to send your message:

```
$ mail cris<CR>
The meeting has been changed to 2:00.
.<CR>
mail: Can't send to cris
mail: Return to kol
you have mail in /usr/mail/kol
$
```

The mail that is waiting for you in **/usr/mail** will be useful if you do not know why the **mail** command has failed, or if you want to retrieve your mail so that you can resend it without typing it in again. It contains the following:

```
$ mail<CR>
From kol Sat Jan 18 17:33 EST 1986
>From kol Sat Jan 18 17:33 EST 1986 forwarded by kol
***** UNDELIVERABLE MAIL sent to cris, being returned by marmaduk!kol *****
mail: ERROR # 8 'Invalid recipient' encountered on system marmaduk
The meeting has been changed to 2:00.
?
```

To learn how to display and handle this message, see *Managing Incoming Messages* later in this chapter.

Sending Mail to One Person

The following screen shows a typical message:

```
$ mail tommy<CR>
Tom,<CR>
There's a meeting of the review committee<CR>
at 3:00 this afternoon.  D.F. wants your<CR>
comments and an idea of how long you think<CR>
the project will take to complete.<CR>
B.K.<CR>
.<CR>
$
```

When Tom logs in at his terminal (or while he is already logged in), he receives a message that tells him he has mail waiting:

```
$ you have mail
```

To find out how he can read his mail, see the section *Managing Incoming Messages* in this chapter.

You can practice using the **mail** command by sending mail to yourself. Type in the **mail** command and your login ID, then write a short message to yourself. When you type the final period or <^d>, the mail is sent to a file named after your login ID in the **/usr/mail** directory, and you will receive a notice that you have mail.

Sending mail to yourself can also serve as a handy reminder system. For example, suppose you (login ID **bob**) want to call someone the next morning. Send yourself a reminder in a mail message:

```
$ mail bob<CR>
Call Accounting and find out<CR>
why they haven't returned my 1988 figures!<CR>
.<CR>
$
```

When you log in the next day, a notice appears on your screen informing you that you have mail waiting to be read.

Sending Mail to Several People Simultaneously

You can send a message to a number of people by including their login names on the **mail** command line. For example:

```
$ mail tommy jane wombat dave<CR>
Diamond cutters,<CR>
The game is on for tonight at diamond three.<CR>
Don't forget your gloves!<CR>
Your Manager<CR>
.<CR>
$
```

Table 8-1 summarizes the syntax and capabilities of the **mail** command.

Table 8-1. Summary of Sending Messages with the **mail** Command

Command Recap **mail** – sends a message to another user's login		
Command	**Options**	**Arguments**
mail	none	*[system_name!]login*
Description:	Typing **mail** followed by one or more login names, sends the message typed on the lines following the command line to the specified login(s).	
Remarks:	Typing a period or a <^d> (followed by the RETURN key) at the beginning of a new line sends the message.	

Identifying Remote Systems: uname and uuname

Until now we have assumed that you are sending messages to users on the local operating system. However, your company may have three separate computer systems, each in a different part of a building, or you may have offices in several locations, each with its own system.

You can send mail to users on other systems by adding the name of the recipients system before the login ID on the command line:

mail sys2!bob<CR>

Notice that the system name and the recipient's login ID are separated by an exclamation point.

Before you can run this command, however, you need three pieces of information:

1. the name of the remote system
2. whether or not your system and the remote system communicate
3. the recipients login name

The **uname** and **uuname** commands allow you to find this information.

If you can, get the name of the remote system and the recipients login name from the recipient. If the recipient does not know the system name, have him or her issue the following command on the remote system:

uname –n<CR>

The command responds with the name of the system. For example:

```
$ uname –n<CR>
writer
$
```

Once you know the remote system name, the **uuname** command can help you verify that your system can communicate with the remote system. At the prompt, type:

uuname<CR>

This generates a list of remote systems with which your system can communicate. If the recipients system is on that list, you can send messages to it by **mail**.

You can simplify this step by using the **grep** command to search through the **uuname** output. At the prompt, type:

uuname | grep *system*<CR>

(Here *system* is the recipients system name.) If **grep** finds the specified system name, it prints it on the screen. For example:

```
$ uuname | grep writer<CR>
writer
$
```

This means that **writer** can communicate with your system. If **writer** does not communicate with your system, **uuname** returns a prompt.

```
$ uuname | grep writer<CR>
$
```

To summarize our discussion of **uname** and **uuname,** consider an example. Suppose you want to send a message to login **sarah** on the remote system **writer.** Verify that **writer** can communicate with your system and send your message. The following screen shows both steps:


```
$ uuname | grep writer<CR>
writer
$ mail writer!sarah<CR>
Sarah,<CR>
The final counts for the writing seminar<CR>
are as follows:<CR>
<CR>
Our department – 18<CR>
Your department – 20<CR>
<CR>
Tom<CR>
.<CR>
$
```

Tables 8-2 and 8-3 summarize the syntax and capabilities of the **uname** and **uuname** commands, respectively.

Table 8-2. Summary of the **uname** Command

Command Recap **uname** – displays the system name		
Command	**Options**	**Arguments**
uname	**–n** and others*	none
Description:	**uname –n** displays the name of the system on which your login resides.	

* See the **uname**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Table 8-3. Summary of the **uuname** Command

Command Recap **uuname** – displays a list of networked systems		
Command	**Options**	**Arguments**
uuname	available	none
Description:	**uuname** displays a list of remote systems that can communicate with your system.	

* See the **uuname**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabiblities.

Managing Incoming Messages

As stated earlier, the **mail** command also allows you to display messages sent to you by other users on your screen so you can read them. If you are logged in when someone sends you mail, the following message is printed on your screen:

```
you have mail
```

NOTE

This notification is controlled by the **MAIL**, **MAILPATH**, and **MAILCHECK** shell variables described in **sh**(1) of the *User's Reference Manual*.

This means that one or more messages are being held for you in a file called **/usr/mail/***your_login*, usually referred to as your mailbox. To display these messages on your screen, type the **mail** command without any arguments:

mail<CR>

The messages display one at a time, beginning with the one most recently received. A typical **mail** message display looks like this:

```
$ mail
From tommy Wed May 21 15:33 CST 1986
Bob,
Looks like the meeting has been cancelled.
Do you still want the material for the technical review?
Tom

?
```

The first line, called the header, provides information about the message: the login name of the sender and the date and time the message was sent. The lines after the header (up to the line containing the ?) comprise the text of the message.

If a long message is being displayed on your terminal screen, you may not be able to read it all at once. You can interrupt the printing by typing <^**s**>. This will freeze the screen, giving you a chance to read. When you are ready to continue, type <^**q**> and the printing will resume.

After displaying each message, the **mail** command prints a **?** prompt and waits for a response. You have many options; for example, you can leave the current message in your mailbox while you read the next message; you can delete the current message; or you can save the current message for future reference. For a list of **mail**s available options, type a **?** in response to **mail**s **?** prompt.

To display the next message without deleting the current message, press the **RETURN** key after the question mark.

?<CR>

The current message remains in your mailbox and the next message displays. If you have read all the messages in your mailbox, a prompt appears.

To delete a message, type a **d** after the question mark:

```
? d<CR>
```

The message is deleted from your mailbox. If there is another message waiting, it then displays. If not, a prompt appears as a signal that you have finished reading your messages.

To save a message for later reference, type an **s** after the question mark:

```
? s<CR>
```

This saves the message, by default, in a file called **mbox** in your home directory. To save the message in another file, type the name of that file after the **s** command.

For example, to save a message in a file called **mailsave** (in your current directory), enter the response shown after the question mark:

```
? s mailsave<CR>
```

If **mailsave** is an existing file, the **mail** command appends the message to it. If there is no file by that name, the **mail** command creates one and stores your message in it. You can later verify the existence of the new file by using the **ls** command. (**ls** lists the contents of your current directory.)

You can also save the message in a file in a different directory by specifying a path name. For example:

```
? s project1/memo<CR>
```

This is a relative path name that identifies a file called **memo** (where your message will be saved) in a subdirectory (**project1**) of your current directory. You can use either relative or full path names when saving mail messages. (For instructions on using path names, see Chapter 3.)

To quit reading messages, enter the response shown after the question mark:

```
? q<CR>
```

Any messages that you have not read are kept in your mailbox until the next time you use the **mail** command.

To stop the printing of a message entirely, press the BREAK key. The **mail** command stops the display, print a ? prompt, and waits for a response from you.

Table 8-4 summarizes the syntax and capabilities of the **mail** command for reading messages.

Table 8-4. Summary of Reading Messages with the **mail** Command

Command Recap **mail** – reads messages sent to your login		
Command	**Options**	**Arguments**
mail	available*	none
Description:	When issued without options, the **mail** command displays any messages waiting in your mailbox (system file **/usr/mail/***your_login*).	
Remarks:	A question mark (**?**) at the end of a message means that a response is expected. A full list of possible responses is given in the *User's Reference Manual*.	

* See the **mail**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

The mailx Facility

This section introduces the **mailx** facility. It explains how to set up your **mailx** environment, send messages with the **mailx** command, and handle messages that have been sent to you. The material is presented in four parts:

- This overview
- Sending Messages
- Managing Incoming Messages
- The **.mailrc** File

The **mailx** command is an enhanced version of the **mail** command. There are many options to **mailx** that are not available in **mail** for sending and reading mail. For example, you can define an alias for a single login or for a group. This allows you to send **mail** to an individual using a name or word other than their login ID, and to send **mail** to a whole group of people using a single name or word. When you use **mailx** to read incoming mail, you can save it in various files, edit it, forward it to someone else, respond to the person who originated the message. By using **mailx** environment variables, you can develop an environment to suit your individual tastes.

If you type the **mailx** command with one or more logins as arguments, **mailx** decides you are sending mail to the named users, prompts you for a summary of the subject, and then waits for you to type in your message or issue a command. The section *Sending Messages* describes features that are available to you for editing, incorporating other files, adding names to copy lists, and more.

If you enter the **mailx** command with no arguments, **mailx** checks incoming mail for you in a file named **/usr/mail/***your_login*. If there is mail for you in that file, you are shown a list of the items and given the opportunity to read, store, remove or transfer each one to another file. The section entitled *Managing Incoming Messages* provides some examples and describes the options available.

If you choose to customize **mailx**, you should create a start-up file in your home directory called **.mailrc**. The section on *The .mailc File* describes variables you can include in your start-up file.

mailx has two modes of functioning: input mode and command mode. You must be in input mode to create and send messages. Command mode is used to read incoming mail. You can use any of the following methods to control the way **mailx** works for you:

- by entering options on the command line. (See the **mailx**(1) manual page in the *User's Reference Manual*.)
- by issuing commands when you are in input mode, for example, creating a message to send. These commands are always preceded by a ~ (tilde) and are referred to as tilde escapes. (See the **mailx**(1) manual page in the *User's Reference Manual*.)
- by issuing commands when you are in command mode, for example, reading incoming mail.
- by storing commands and environment variables in a start-up file in your home directory called **$HOME/.mailrc**.

Tilde escapes are discussed in *Sending Messages: Tilde Escapes*, command mode commands in *Managing Incoming Messages*; and the **.mailrc** file in *The .mailrc File*.

Command Line Options

This section discusses command line options. The syntax for the **mailx** command is:

mailx [*options*] [*name...*]

The *options* are flags that control the action of the command, and *name...* represents the intended recipients.

Anything on the command line other than an option preceded by a hyphen is read by **mailx** as a *name*; i.e., the login or alias of a person to whom you are sending a message.

Two of the command line options deserve special mention:

–f

Allows you to read messages from *filename* instead of your mailbox.

Because **mailx** lets you store messages in any file you name, you need the **–f** option to review these stored options. The default storage file is **$HOME/mbox**, so use the following command to review messages:

```
mailx –f
```

–n:

Do not initialize from the system default **mailx.rc** file.

If you have your own **.mailrc** file (see *The .mailrc File*), **mailx** does not look through the system default file for specifications when you use the **–n** option, but goes directly to your **.mailrc** file. This results in faster initialization; substantially faster when the system is busy.

Sending Messages: Tilde Escapes

To send a message to another operating system user, enter the following command:

```
$ mailx daves<CR>
```

The login name specified belongs to the person who is to receive the message. The system puts you into input mode and prompts you for the subject of the message. (You may have to wait a few seconds for the `Subject:` prompt if the system is very busy.) This is the simplest way to run the **mailx** command; it differs very little from the way you run the **mail** command.

The following examples show how you can edit messages you are sending, incorporate existing text into your messages, change the header information, and perform other tasks that take advantage of the **mailx** commands capabilities. Each example is followed by an explanation of the key points illustrated in the example.

```
$ mailx daves<CR>
Subject:
```

Whether to include a subject or not is optional. If you elect not to, press the RETURN key. The cursor moves to the next line and the program waits for you to enter the text of the message.

```
$ mailx daves<CR>
Subject: meeting<CR>
We're having a meeting for novice mailx users in<CR>
the auditorium at 9:00 tomorrow.<CR>
Would you be willing to give a demonstration?<CR>
Bob<CR>
~. <CR>
EOT
$
```

There are two important things to notice about the above example:

- You break up the lines of your message by pressing the RETURN key at the end of each line. This makes it easier for the recipient to read the message, and prevents you from overflowing the line buffer.
- You end the text and send the message by entering a tilde and a period together (~.) at the beginning of a line. The system responds with an end-of-text (EOT) notice and a prompt.

There are several commands available to you when you are in input mode (you were in input mode in the example). Each of them consists of a tilde (~), followed by an alphabetic character, entered at the beginning of a line. Together they are known as tilde escapes. (See the **mailx**(1) manual page in the *User's Reference Manual*.) Most of them are used in the examples in this section.

You can include the subject of your message on the command line by using the **–s** option. For example, the command line:

```
$ mailx –s "meeting" daves<CR>
```

is equivalent to:

```
$ mailx daves<CR>
Subject: meeting<CR>
```

The subject line looks the same to the recipient of the message. Notice that when putting the subject on the command line, you must enclose a subject that has more than one word in quotation marks.

Editing the Message

When you are in the input mode of **mailx**, you can invoke an editor by entering the ~**e** (tilde e) escape at the beginning of a line. The following example shows how to use tilde:

```
$ mailx daves<CR>
Subject: Testing my tilde<CR>
When entering the text of a message<CR>
that has somehow gotten garbled<CR>
you may invoke your favorite editor<CR>
by means of a ~e (tilde e).
.
.
.
```

Notice that you have misspelled a word in your message. To correct the error, use ~**e** to invoke the editor, in this case the default editor, **ed**.

```
.
.
.
~e<CR>
12
/grabled/p
that has somehow gotten grabled
s/gra/gar/p
that has somehow gotten garbled
w
132
q
(continue)
What more can I tell you?
.
.
.
```

In this example the **ed** editor was used. Your **.profile** or a **.mailrc** file controls which editor is invoked when you issue a ~**e** escape command. The ~**v** (tilde v) escape invokes an alternate editor (most commonly, **vi**).

When you exited from **ed** (by typing **q**), the **mailx** command returned you to input mode and prompted you to continue your message. At this point you may want to preview your corrected message by entering a **~p** (tilde p) escape. The **~p** escape prints out the entire message up to the point where the **~p** was entered. Thus, at any time during text entry, you can review the current contents of your message.

```
~p
Message contains:
To: daves
Subject: Testing my tilde

When entering the text of a message
that has somehow gotten garbled
you may invoke your favorite editor
by means of a tilde e (~e).
What more can I tell you?
(continue)
~.
EOT
$
```

Incorporating Existing Text into Your Message

mailx provides four ways to incorporate material from another source into the message you are creating:

- read a file into your message
- read a message you have received into a reply
- incorporate the value of a named environment variable into a message
- execute a shell command and incorporate the output of the command into a message

The following examples show the first two of these functions. These are the most commonly used of these four functions. For information about the other two, see the **mailx**(1) manual page of the *User's Reference Manual.*

Reading a File into a Message

```
$ mailx daves<CR>
Subject: Work Schedule<CR>
As you can see from the following<CR>
~r letters/file1
"letters/file1"    10/725
we have our work cut out for us.
Please give me your thoughts on this.
– Bob
~.
EOT
$
```

As the example shows, the **~r** (tilde r) escape is followed by the name of the file you want to include. The system displays the file name and the number of lines and characters it contains. You are still in input mode and can continue with the rest of the message. When the recipient gets the message, the text of **letters/file1** is included. (You can, of course, use the **~p** (tilde p) escape to preview the contents before sending your message.)

Incorporating a Message from Your Mailbox into a Reply

```
$ mailx<CR>
mailx version 2.14  2/9/85  Type ? for help.
"usr/mail/roberts": 2 messages 1 new
>N   1 abc         Tue May 1  08:09  8/155  Meeting Notice
     2 hqtrs       Mon Apr 30 16:57  4/127  Schedule
? m jones<CR>
Subject: Hq Schedule<CR>
Here is a copy of the schedule from headquarters...<CR>
~f 2<CR>
Interpolating: 2
(continue)
As you can see, the boss will be visiting our district on<CR>
the 14th and 15th.<CR>
– Robert
~.
EOT
?
```

There are several important points illustrated in this example:

- The sequence begins in command mode, where you read and respond to your incoming mail. Then you switch into input mode by issuing the command **m jones** (meaning send a message to jones).
- The **~f** escape is used in input mode to call in one of the messages in your mailbox and make it part of the outgoing message. The number **2** after the **~f** means message 2 is to be interpolated (read in).
- **mailx** tells you that message 2 is being interpolated and then tells you to continue.
- When you finish creating and sending the message, you are back in command mode, as shown by the **?** prompt. You may now do something else in command mode, or exit **mailx** by typing **q**.

An alternate command, the ~**m** (tilde m) escape, works the way that ~**f** does except the read-in message is indented one tab stop. Both the ~**m** and ~**f** commands work only if you start out in command mode and then enter a command that puts you into input mode. Other commands that work this way are covered in the section *Managing Incoming Messages*.

Changing Parts of the Message Header

The header of a **mailx** message has four components:

- subject
- recipient(s)
- copy-to list
- blind-copy list (a list of intended recipients that is not shown on the copies sent to other recipients)

When you enter the **mailx** command followed by a login or an alias, you are put into input mode and prompted for the subject of your message. Once you end the subject line by pressing the RETURN key, **mailx** expects you to type the text of the message. If, at any point in input mode, you want to change or supplement some of the header information, there are four tilde escapes that you can use: ~**h**, ~**t**, ~**c**, and ~**b**.

~**h** displays all header fields: subject, recipient, copy-to list, and blind copy list, with their current values. You can change a current value, add to it, or, by pressing the RETURN key, accept it.

~**t** lets you add names to the list of recipients. Names can be either login names or aliases.

~**c** lets you create or add to a copy-to list for the message. Enter either login names or aliases of those to whom a copy of the message should be sent.

~**b** lets you create or add to a blind-copy list for the message.

All tilde escapes must be in the first position on a line. For the ~**t**, ~**c** or ~**b**, any additional material on the line is taken to be input for the list in question. Any additional material on a line that begins with a ~**h** is ignored.

Adding Your Signature

If you want, you can establish two different signatures with the **sign** and **Sign** environment variables. These can be invoked with the ~**a** (tilde a) or ~**A** (tilde A) escape, respectively. In the following example, you have set the value Supreme Commander to be called by the ~**A** escape:

```
$ mailx -s orders all<CR>
Be ready to move out at 0400 hours.<CR>
~A<CR>
Supreme Commander
~.<CR>
EOT
$
```

Having both escapes (~**a** and ~**A**) allows you to set up two forms for your signature. However, because the senders login automatically appears in the message header when the message is read, no signature is required to identify you.

Keeping a Record of Messages You Send

The **mailx** command offers several ways to keep copies of outgoing messages. Two that you can use without setting any special environment variables are the ~**w** (tilde w) escape and the **-F** option on the command line.

The ~**w** followed by a file name causes the message to be written to the named file. For example:

```
$ mailx bdr<CR>
Subject: Saving Copies<CR>
When you want to save a copy of<CR>
the text of a message, use the tilde w.<CR>
~w savemail
"savemail" 2/71
~.
EOT
$
```

If you now display the contents of **savemail**, you see:

```
$ cat savemail<CR>
When you want to save a copy of
the text of a message, use the tilde w.
$
```

The drawback to this method, as you can see, is that none of the header information is saved.

Using the **–F** option on the command line preserves the header information:

```
$ mailx –F –s Savings bdr<CR>
This method appends this message to a
file in my current directory named bdr.
~.
EOT
$
```

Check the results by looking at the file **bdr**:

```
$ cat bdr<CR>
From: kol  Fri May 2  11:14:45  1986
To: bdr
Subject: Savings

This method appends this message to a
file in my current directory named bdr.
$
```

The **–F** option appends the text of the message to a file named after the first recipient. If you have used an alias for the recipient(s), the alias is first converted into the appropriate login(s) and the first login is used as the file name. As noted above, if you have a file by that name in your current directory, the text of the message is appended to it.

Exiting from mailx

When you have finished composing your message, you can leave **mailx** by typing any of the following three commands:

~. tilde period (**~.**) is the standard way of leaving input mode. It also sends the message. If you entered input mode from the command mode of **mailx**, you now return to the command mode (as shown by the ? prompt you receive after typing this command). If you started out in input mode, you now return to the shell (as shown by the shell prompt).

~q tilde q (**~q**) simulates an interrupt. It lets you exit the input mode of **mailx**. If you have entered text for a message, it is saved in a file called **dead.letter** in your home directory.

~x tilde x (**~x**) simulates an interrupt. It lets you exit the input mode of **mailx** without saving anything.

Managing Incoming Messages

mailx has over fifty commands to help you manage your incoming mail. See the **mailx**(1) manual page in the *User's Reference Manual* for a list of all of them (and their synonyms) in alphabetic order. The most commonly used commands (and arguments) are described in the following subsections:

- the *msglist* argument
- commands for reading and deleting mail
- commands for saving mail
- commands for replying to mail
- commands for getting out of **mailx**

msglist Argument

Many commands in **mailx** take a form of the *msglist* argument. This argument provides the command with a list of messages on which to operate. If a command expects a *msglist* argument and you do not provide one, the command is performed on the current message. Any of the following formats can be used for a *msglist*:

n
message number *n*, the current message

^
the first undeleted message

$
the last message

*
all messages

n-m
an inclusive range of message numbers

user
all messages from *user*

/string
All messages with *string* in the subject line (case is ignored)

:c
all messages of type *c* where *c* is:

d - deleted messages
n - new messages
o - old messages
r - read messages
u - unread messages

The context of the command determines whether this type of specification makes sense.

Here are two examples (the ? is the command mode prompt):

```
? d 1-3          [ Delete messages 1, 2 and 3 ]
? s bdr bdr      [ Save all messages from user bdr in a
                   file named bdr. ]

?
```

Additional examples may be found throughout the next three subsections.

Reading and Deleting Mail

When a message arrives in your mailbox the following notice appears on your screen:

```
you have mail
```

The notice appears when you log in or when you return to the shell from another procedure. (This notice is controlled by the **MAILCHECK** shell variable. See **sh**(1) in the *User's Reference Manual*.)

Reading Mail

To read your mail, enter the **mailx** command with or without arguments. Execution of the command places you in the command mode of **mailx**. The next thing that appears on your screen is a display that looks something like this:

```
mailx version 2.14  10/19/86    Type ? for help
"/usr/mail/bdr":   3 messages  3 new
> N 1 rbt          Thur Apr 30 14:20   8/190  Review Session
  N 2 admin        Thur Apr 30 15:56   5/84   New printer
  N 3 daves        Fri  May  1 08:39  64/1574 Reorganization
?
```

The first line identifies the version of **mailx** used on your system, displays the date, and reminds you that help is available by typing a question mark (**?**). The second line shows the path name of the file used as input to the display (the file name is normally the same as your login name) together with a count of the total number of messages and their status. The rest of the display is header information from the incoming messages.

The messages are numbered in sequence with the last one received at the bottom of the list. To the left of the numbers there may be a status indicator; N for new, U for unread. A greater than sign (**>**) points to the current message. Other fields in the header line show the login of the originator of the message, the day, date and time it was delivered, the number of lines and characters in the message, and the message subject. The last field may be blank.

When the header information displays on your screen, you can print messages either by pressing the **RETURN** key or entering a command followed by a *msglist* argument. If you enter a command with no *msglist* argument, the command acts on the message pointed at by the **>** sign. Pressing the **RETURN** key is the equivalent of a typing the **p** (for print) command without a *msglist* argument; the message displayed is the one pointed at by the **>** sign. To read some other message (or several others in succession), enter a **p** (for print) or **t** (for type) followed by the message number(s).

Here are some examples:

```
? <CR>          [ Print the current message. ]
? p 2<CR>       [ Print message number 2.    ]
? p daves<CR>   [ Print all messages from user daves. ]
```

The command **t** (for type) is a synonym of **p** (for print).

Scanning Your Mailbox

The **mailx** command lets you look through the messages in your mailbox while you decide which ones need your immediate attention.

When you first enter the **mailx** command mode, the banner tells you how many messages you have and displays the header line for 20 messages. (If you are dialed into the computer system, only the header lines for 10 messages display.) If the total number of messages exceeds one screenful, you can display the next screen by entering the **z** command. Typing **z–** causes a previous screen (if there is one) to display. If you want to see the header information for a specific group of messages, enter the **f** (for from) command followed by the *msglist* argument.

Here are examples of those commands:

```
? z              [ Scroll forward one screenful of header lines. ]
? z-             [ Scroll backward one screenful. ]
? f daves        [ Display headers of all messages from user daves. ]
```

Switching to Other Mail Files

When you enter **mailx** by issuing the following command, you are looking at the file **/usr/mail/***your_login*:

```
$ mailx<CR>
```

mailx lets you switch to other mail files and use any of the **mailx** commands on their contents. (You can even switch to a non-mail file, but if you try to use **mailx** commands, you are told `No applicable messages`.) The switch to another file is done with the **fi** or **fold** command (they are synonyms) followed by the *filename*. The following special characters work in place of the *filename* argument:

%	the current mailbox
%*login*	the mailbox of the owner of *login* (if you have the required permissions)
#	the previous file
&	the current mbox

Here is an example of how this might look on your screen:

```
$ mailx<CR>

mailx version 2.14 10/19/86  Type ? for help.
"usr/mail/daves":  3 messages 2 new 3 unread
  U 1 jaf          Sat May 9 07:55   7/137   test25
> N 2 todd         Sat May 9 08:59   9/377   UNITS requirements
  N 3 has          Sat May 9 11:08  29/1214  access to bailey

? fi &            [ Enter this command to transfer to your mbox. ]

Held 3 messages in /usr/mail/daves
"/fs1/daves/mbox":  74 messages 10 unread
   .
   .
   .
? q<CR>
$
```

Deleting Mail

To delete a message, enter a **d** followed by a *msglist* argument. If the *msglist* argument is omitted, the current message is deleted. The messages are not deleted until you leave the mailbox file you are processing. Prior to that, the **u** (for undelete) gives you the opportunity to change your mind. Once you have issued the quit command (**q**) or switched to another file, however, the deleted messages are gone.

mailx permits you to combine the delete and print command and enter a **dp**. This is like saying, "Delete the message I just read and show me the next one."

Here are some examples of the delete command:

```
? d *     [ Delete all my messages. ]
? d r     [ Delete all messages that have been read. ]
? dp      [ Delete the current message and print the next one. ]
? d 2-5   [ Delete messages 2 through 5. ]
```

Saving Mail

All messages not specifically deleted are saved when you quit **mailx**. Messages that have been read are saved in a file in your home directory called **mbox**. Messages that have not been read are held in your mailbox (**/usr/mail/***your_login*).

The command to save messages comes in two forms: with an uppercase or a lowercase **s**. The syntax for the uppercase version is:

S [*msglist*]

Messages specified by the *msglist* argument are saved in a file in the current directory named for the author of the first message in the list.

The syntax for the lowercase version is:

s [*msglist*] [*filename*]

Messages specified by the *msglist* argument are saved in the file named in the *filename* argument. If you omit the *msglist* argument, the current message is saved. If you are using logins for file names, this can lead to some ambiguity. If **mailx** is puzzled, you get an error message.

Replying to Mail

The command for replying to mail comes in two forms: an uppercase or a lowercase **r**. The principal difference between the two forms is that the uppercase form (**R**) causes your response to be sent only to the originator of the message, while the lowercase form (**r**) causes your response to be sent not only to the originator but also to all other recipients. (There are other differences between these two forms. For details, see the **mailx**(1) manual page in the *User's Reference Manual*.)

When you reply to a message, the original subject line is picked up and used as the subject of your reply. Here's an example of the way it looks:

```
$ mailx<CR>

mailx version 2.14 10/19/83  Type ? for help.
"usr/mail/daves":  3 messages 2 new 3 unread
  U 1 jaf          Wed May 9 07:55   7/137   test25
> N 2 todd         Wed May 9 08:59   9/377   UNITS requirements
  N 3 has          Wed May 9 11:08  29/1214  access to bailey

? R 2
To: todd
Subject: Re: UNITS requirements
```

Assuming the message about UNITS requirements was sent to other users, and the lowercase **r** had been used, the header might have appeared like this:

```
? r 2
To: todd eg has jcb bdr
Subject: Re: UNITS requirements
```

Leaving mailx

There are two standard ways of leaving **mailx**: with a **q** or an **x**. If you leave **mailx** with a **q**, you see messages that summarize what you did with your mail. They look like this:

```
? q<CR>
Saved 1 message in /fs1/bdr/mbox
Held 1 message in /usr/mail/bdr
$
```

From the example, you can surmise that user **bdr** had at least two messages, read one and either left the other unread or issued a command asking to hold it in **/usr/mail/bdr**. If there were more than two messages, the others were deleted or saved in other files. **mailx** does not issue a message about those messages.

If you leave **mailx** with an **x**, it is almost as if you had never entered. Mail read and messages deleted are retained in your mailbox. However, if you have saved messages in other files, that action has already taken place and is not undone by the **x**.

mailx Online Help

The preceding subsections described some of the most frequently used **mailx** commands. (See the **mailx**(1) manual page in the *User's Reference Manual* for a complete list.) If you need help while you are in the command mode of **mailx**, type either a **?** or **help** after the **?** prompt. A list of **mailx** commands and what they do displays on your terminal screen.

The .mailrc File

The **.mailrc** file contains commands to be executed when you invoke **mailx**.

There may be a system-wide start-up file (**/usr/lib/mailx/mailx.rc**) on your system. If it exists, it is used by the System Administrator to set common variables. Variables set in your **.mailrc** file take precedence over those in **mailx.rc.**

Most **mailx** commands are legal in the **.mailrc** file. However, the following commands are *not* legal entries:

! (or) **shell**
escape to the shell

Copy
save messages in *msglist* in a file whose name is derived from the author

edit
invoke the editor

visual
invoke **vi**

followup
respond to a message

Followup
respond to a message, sending a copy to *msglist*

mail
switch into input mode

reply
respond to a message

Reply
respond to the author of each message in *msglist*

You can create your own **.mailrc** with any editor, or copy another users. Figure 8-1 shows a sample **.mailrc** file.

```
if r
        cd $HOME/mail
endif
set allnet append asksub askcc autoprint dot
set metoo quiet save showto header hold keep keepsave
set outfolder
set folder='mail'
set record='outbox'
set crt=24
set EDITOR='/bin/ed'
set sign='Roberts'
set Sign='Jackson Roberts, Supervisor'
set toplines=10
alias fred              fjs
alias bob               rcm
alias alice             ap
alias mark              mct
alias donna             dr
alias pat               pat
group robertsgrp        fred bob alice pat mark
group accounts          robertsgrp donna
```

Figure 8-1. Sample **.mailrc** File

The example in Figure 8-1 includes the commands you are most likely to find useful: the **set** command and the **alias** or **group** commands.

The **set** command is used to establish values for environment variables. The command syntax is:

set
set *name*
set *name* = *string*
set *name* = *number*

When you issue the **set** command without any arguments, **set** produces a list of all defined variables and their values. The argument *name* refers to an environmental variable. More than one *name* can be entered after the **set** command. Some variables take a string or numeric value; string values are enclosed in single quotes.

When you put a value in an environment variable operating systemn assignment such as **HOME**=*my_login*, you are telling the shell how to interpret that variable. However, this type of assignment in the shell does not make the value of the variable accessible to other operating system programs that need to reference environment variables. To make it accessible, you must export the variable. If you set the **TERM** variable in your environment in Chapter 6 or Chapter 7, you will remember using the **export** command as shown in the following example:

```
$ TERM=vt100
$ export TERM
```

When you export variables from the shell in this way, programs that reference environment variables are said to import them. Some of these variables (e.g., **EDITOR** and **VISUAL**) are not peculiar to **mailx**, but may be specified as general environment variables and imported from your execution environment. If a value is set in **.mailrc** for an imported variable, it overrides the imported value. There is an **unset** command, but it works only against variables set in **.mailrc**; it has no effect on imported variables.

There are 41 environment variables that can be defined in your **.mailrc**; too many to be fully described in this document. For complete information, see the **mailx**(1) manual page in the *User's Reference Manual*.

Three variables used in the example in Figure 8-1 deserve special attention because they demonstrate how to organize the filing of messages. These variables are: **folder**, **record**, and **outfolder**. All three are interrelated and control the directories and files in which copies of messages are kept.

To put a value into the **folder** variable, use the following format:

set folder=*directory*

This specifies the directory in which you want to save standard mail files. If the directory name specified does not begin with a **/** (slash), it is presumed to be relative to **$HOME**. If **folder** is an exported shell variable, you can specify file names (in commands that call for a *filename* argument) with a **/** before the name; the name is expanded so that the file is put into the **folder** directory.

To put a value in the **record** variable, use the following format:

set record=*filename*

This directs **mailx** to save a copy of all outgoing messages in the specified file. The header information is saved with the text of the message. By default, this variable is disabled.

The **outfolder** variable causes the file in which you store copies of outgoing messages (enabled by the variable **record**=) to be located in the **folder** directory. It is established by being named in a **set** command. The default is **nooutfolder**.

The **alias** and **group** commands are synonyms. In Figure 8-1, the **alias** command is used to associate a name with a single login; the **group** command is used to specify multiple names that can be called in with one pseudonym. This is a nice way to distinguish between single and group aliases, but if you want, you can treat the commands as exact equivalents. Notice, too, that aliases can be nested.

In the **.mailrc** file shown in Figure 8-1, the alias **robertsgroup** represents five users; three of them are specified by previously defined aliases and one is specified by a login. The fifth user, **pat**, is specified by both a login and an alias. The next group command in the example, **accounts**, uses the alias **robertsgroup** plus the alias **donna**. It expands to 12 logins.

The **.mailrc** file in Figure 8-1 includes an **if-endif** command. The full syntax of that command is:

if s | **r** *mail_commands*

else *mail_commands*

endif

The **s** and **r** stand for send and receive, so you can cause some initializing commands to be executed according to whether **mailx** is entered in input mode (send) or command mode (receive). In the preceding example, the command is issued to change directory to **$HOME/mail** if reading mail. The user in this case had elected to set up a subdirectory for handling incoming mail.

The environment variables shown in this section are those most commonly included in the **.mailrc** file. You can, however, specify any of them for one session only whenever you are in command mode. For a complete list of the environment variables you can set in **mailx**, see the **mailx**(1) manual page in the *User's Reference Manual.*

Sending and Receiving Files

This section describes the commands available for transferring files: the **mail** command for small files (a page or less), and the **uucp** and **uuto** commands for long files. The **mail** command can be used to transfer file either within a local system or to a remote system. The **uucp** and **uuto** commands transfer files from one system to another.

Sending Small Files: mail Command

To send a file in a **mail** message, you must redirect the input to that file on the command line. Use the < (less than) redirection symbol as follows:

mail *login* < *filename*<**CR**>

(For further information on input redirection, see Chapter 7.) Here *login* is the recipients login ID and *filename* is the name of the file you want to send. For example, to send a copy of a file called **agenda** to the owner of login **sarah** (on your system), type the following command line:

```
$ mail sarah < agenda<CR>
$
```

The prompt that appears on the second line means the contents of **agenda** have been sent. When **sarah** issues the **mail** command to read her messages, she will receive **agenda**.

To send the same file to more than one user on your system, use the same command line format with one difference; in place of one login ID, type several, separated by spaces. For example:

```
$ mail sarah tommy dingo wombat < agenda<CR>
$
```

Again, the prompt returned by the system in response to your command is a signal that your message has been sent.

The same command line format, with one addition, can also be used to send a file to a user on a remote system that can communicate with yours. In this case, you must specify the name of the remote system before the users login name. Separate the system name and the login name with an **!** (exclamation point):

mail *system*!*login* < *filename*<**CR**>

For example:

```
$ mail writer!wombat < agenda<CR>
$
```

The system prompt on the second line means that your message (containing the file) has been queued for sending.

If you are using **mailx**, you cannot use the **mail** command line syntax to send a file. Instead, you use the ~**r** option as follows:

```
mailx phyllis
Subject: Memo
~r memo
$
```

Sending Large Files

The **uucp** and **uuto** commands allow you to transfer files to a remote computer. **uucp** allows you to send files to the directory of your choice on the destination system. If you are transferring a file to a directory that you own, you have permission to put the file in that directory. (See Chapter 3 for information on directory and file permissions.) However, if you are transferring the file to another users directory, you must be sure, in advance, that the user has given you permission to write a file to his or her directory. In addition, because you must specify path names that are often long and accuracy is required, **uucp** command lines may be cumbersome and lead to error.

The **uuto** command is an enhanced version of **uucp**. It automatically sends files to a public directory on the recipients system called **/usr/spool/uucppublic**. This means you cannot choose a destination file. However, it also means that you can transfer a file at any time without having to request write permission from the owner of the destination directory. Finally, the **uuto** command line is shorter and less complicated than the **uucp** command line. When you type a **uuto** command line, the likelihood of making an error is greatly reduced.

Checking Permissions

Before you actually send a file with the **uucp** or **uuto** command, you need to find out whether or not the file is transferable. To do that, you must check the files permissions. If they are not correct, you must use the **chmod** command to change them, if you own the files. (Permissions and the **chmod** command are covered in Chapter 3.)

There are two permission criteria that must be met before a file can be transferred using **uucp** or **uuto**.

- The file to be transferred must have read permission (**r**) for others.
- The directory that contains the file must have read (**r**) and execute (**x**) permission for others.

For example, assume that you have a file named **chicken**, under a directory named **soup** (in your home directory). You want to send a copy of the **chicken** file to another user with the **uuto** command. First, check the permissions on **soup**:

```
$ ls -l<CR>
total 4
drwxr-xr-x    2   reader group1   45   Feb 9   10:43   soup
$
```

The response of the **ls** command shows that **soup** has read (**r**) and execute (**x**) permissions for all three groups; no changes have to be made. Now use the **cd** command to move from your home directory to **soup**, and check the permissions on the file **chicken**.

```
$ ls -l chicken<CR>
total 4
-rw-------    1   reader group1 3101   Mar 1   18:22   chicken
$
```

The command's output means that you (the user) have permission to read the file **chicken**, but no one else does. To add read permissions for your group (**g**) and others (**o**), use the **chmod** command:

```
$ chmod go+r chicken<CR>
```

Now check the permissions again with the **ls –l** command:

```
$ ls –l chicken<CR>
total 4
-rw-r--r--     1   reader group1 3101    Mar01   18:22   chicken
$
```

This confirms that the file is now transferable; you can send it with the **uucp** or **uuto** command. After you send copies of the file, you can reverse the procedure and replace the previous permissions.

uucp Command

The command **uucp** allows you to copy a file directly to the home directory of a user on another computer, or to any other directory you specify and for which you have write permission.

uucp is not an interactive command. It performs its work silently, invisible to the user. Once you issue this command you may run other processes.

Transferring a file between computers is a multiple-step procedure. First, a work file, containing instructions for the file transfer, must be created. When requested, a data file (a copy of the file being sent) is also made. Then the file is ready to be sent. When you issue the **uucp** command, it performs the preliminary steps described above (creating the necessary files in a dedicated directory called a **spool** directory), then calls the **uucico** daemon that actually transfers the file. (Daemons are system processes that run in background.) The file is placed in a queue and **uucico** sends it at the first available time.

Thus, the **uucp** command allows you to transfer files to a remote computer without knowing anything except the name of the remote computer and, possibly, the login ID of the remote user(s) to whom the file is being sent.

Command Line Syntax

uucp allows you to send:

- one file to a file or a directory
- multiple files to a directory

To deliver your file(s), **uucp** must know the full path name of both the *source-file* and the *destination-file*. However, this does not mean you must type out the full path name of both files every time you use the **uucp** command. There are several abbreviations you can use once you become familiar with their formats; **uucp** expands them to full path names.

To choose the appropriate designations for your *source-file* and *destination-file*, begin by identifying the *source-file*s location relative to your own current location in the file system. (Assume, for the moment, that the *source-file* is in your local system.) If the *source-file* is in your current directory, you can specify it by its name alone (without a path). If the *source-file* is not in your current directory, you must specify its full path name.

How do you specify the *destination-file*? Because it is on a remote system, the *destination-file* must always be specified with a path name that begins with the name of the remote system. After that, however, **uucp** gives you a choice: you can specify the full path or use either of two forms of abbreviation. Your *destination-file* should have one of the following three formats:

- *system_name*!*full_path*
- *system_name*!~*login_name*[/*directory_name*/*filename*]
- *systemname*!~/*login_name*[/*directory_name*/*filename*]

The login name, in this case, belongs to the recipient of the file.

Until now we have described what to do when you want to send a file from your local system to a remote system. However, it is also possible to use **uucp** to send a file from a remote system to your local system. In either case, you can use the formats described above to specify either *source-files* or *destination-files*. The important distinction in choosing one of these formats is not whether a file is a *source-file* or a *destination-file*, but where you are currently located in the file system relative to the files you are specifying. Therefore, in the formats shown above, the *login_name* could refer to the login of the owner or the recipient of either a *source-file* or a *destination-file*.

For example, you are login **kol** on a system called **mickey**. Your home directory is **/usr/kol** and you want to send a file called **chap1** (in a directory called **text** in your home directory) to login **wsm** on a system called **minnie**. You are currently working in **/usr/kol/text**, so you can specify the *source-file* with its relative path name, **chap1**. Specify the *destination-file* in any of the ways shown in the following command lines:

- Specify the *destination-file* with its full path name:

  uucp chap1 minnie!/usr/wsm/receive/chap1

- Specify the *destination-file* with ~*login_name* (which expands to the name of the recipient's home directory) and a name for the new file.

  uucp chap1 minnie!~ wsm/receive/chap1

  (The file will go to **minnie!/usr/wsm/receive/chap1**.)

- Specify the *destination-file* with ~*login_name* (which expands to the recipients home directory) but without a name for the new file; **uucp** gives the new file the same name as the *source-file.*

  uucp chap1 minnie!~ wsm/receive

  (The file will go to **minnie!/usr/wsm/receive/chap1**.)

- Specify the *destination-file* with ~*/login_name.* This expands to the recipients subdirectory in the public directory on the remote system.

  uucp chap1 minnie!~ /wsm

  (The file will go to **minnie!/usr/usr/spool/uucppublic/wsm**.)

Examples of the uucp Command

Suppose you want to send a file called **minutes** to a remote computer named **eagle**. Enter the command line shown in the following screen:

```
$ uucp -m -s status -j minutes eagle!/usr/gws/minutes<CR>
eagleN3f45
$
```

This sends the file **minutes** (located in your current directory on your local computer) to the remote computer **eagle** and places it under the path name **/usr/gws** in a file named **minutes**. When the transfer is complete, the user **gws** on the remote computer is notified by mail.

The **-m** option ensures that you (the sender) are also notified by mail as to whether or not the transfer has succeeded. The **-s** option, followed by the name of the file (**status**), asks the program to put a status report of the file transfer in the specified file (**status**).

NOTE

Be sure to include a file name after the **-s** option. If you do not, you get the message: **uucp failed completely**.

The job ID (**eagleN3f45**) displays in response to the **–j** option.

Even if **uucp** does not notify you of a successful transfer soon after you send a file, do not assume that the transfer has failed. Not all systems equipped with networking software have the hardware needed to call other systems. Files being transferred from these so called passive systems must be collected periodically by active systems equipped with the required hardware (see *Operation of the uucp Command* for details). Therefore, if you are transferring files from a passive system, you may experience some delay. Check with your System Administrator to find out whether your system is active or passive.

The previous example uses a full path name to specify the *destination-file*. There are two other ways the *destination-file* can be specified:

- The login directory of **gws** can be specified through use of the ~ (tilde), as shown below:

  eagle!~gws/minutes

  is interpreted as:

  eagle!/usr/gws/minutes

- The **uucppublic** area is referenced by a similar use of the tilde prefix to the path name. For example:

  eagle!~/gws/minutes

  is interpreted as:

  /usr/spool/uucppublic/gws/minutes

Operation of the uucp Command

This section is an overview of what happens when you issue the **uucp** command. An understanding of the processes involved may help you to be aware of its limitations and requirements: why it can perform some tasks and not others, why it performs tasks when it does, and why you may or may not be able to use it for tasks that **uucp** performs. For further details, see the *System Administrator's Guide* and the *System Administrator's Reference Manual*.

When you enter a **uucp** command, the **uucp** program creates a work file and usually a data file for the requested transfer. (**uucp** does not create a data file when you use the **–c** option.) The work file contains information required to transfer the file(s). The data file is a copy of the specified source file. After these files are created in the spool directory, the **uucico** daemon is started.

The **uucico** daemon attempts to establish a connection to the remote computer that is to receive the file(s). It first gathers the information required for establishing a link to the remote computer from the **Systems** file. This is how **uucico** knows what type of device to use in establishing the link. Then **uucico** searches the **Devices** file looking for the devices that match the requirements listed in the **Systems** file. After **uucico** finds an available device, it attempts to establish the link and log in on the remote computer.

When **uucico** logs in on the remote computer, it starts the **uucico** daemon on the remote computer. The two **uucico** daemons then negotiate the line protocol to be used in the file transfer(s). The local **uucico** daemon transfers the file(s) that you are sending to the remote computer; the remote **uucico** places the file in the specified path name(s) on the remote computer. After your local computer completes the transfer(s), the remote computer may send files that are queued for your local computer. The remote computer can be denied permission to transfer these files with an entry in the **Permissions** file. If this is done, the remote computer must establish a link to your local computer to perform the transfers.

If the remote computer or the device selected to make the connection to the remote computer is unavailable, the request remains queued in the spool directory. Each hour (default), **uudemon.hour** is started by **cron** which in turn starts the **uusched** daemon. When the **uusched** daemon starts, it searches the spool directory for the remaining work files, generates the random order in which these requests are to be processed, and then starts the transfer process (**uucico**) described in the previous paragraphs.

The transfer process described generally applies to an active computer. An active computer (one with calling hardware and networking software) can be set up to poll a passive computer. Because it has networking software, a passive computer can queue file transfers. However, it cannot call the remote computer because it does not have the required hardware. The **Poll** file (**/usr/lib/uucp/Poll**) contains a list of computers that are to be polled in this manner.

Table 8-5 summarizes the syntax and capabilities of the **uucp** command.

Table 8-5. Summary of the **uucp** Command

Command Recap **uucp** – copies a file from one computer to another		
Command	**Options**	**Arguments**
uucp	**–j1, –m, –s** and others*	*source-file*
Description:	**uucp** performs preliminary tasks required to copy a file from one computer to another, and calls **uucico**, the daemon (background process) that transfers the file. The user need only issue the **uucp** command for a file to be copied.	
Remarks:	By default, the only directory to which you can write files is **/usr/spool/uucppublic**. To write to directories belonging to another user, you must receive write permission from that user. Although there are several ways to represent path names as arguments, it is recommended that you type full path names to avoid confusion.	

* See the **uucp**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

uuto Command

The **uuto** command allows you to transfer files to the public directory of another system. The basic format for the **uuto** command is:

uuto *filename system*!*login*<**CR**>

where *filename* is the name of the file to be sent, *system* is the recipients system, and *login* is the recipients login name.

If you send a file to someone on your local system, you may omit the system name and use the following format:

uuto *filename login*<**CR**>

Sending a File: -m Option and uustat Command

Now that you know how to determine if a file is transferable, this section describes how it works.

The process of sending a file by **uuto** is referred to as a job. When you issue a **uuto** command, your job is not sent immediately. First, the file is stored in a queue (a waiting line of jobs) and assigned a job number. When the jobs number comes up, the file is transmitted to the remote system and placed in a public directory there. The recipient is notified by a **mail** message and must use the **uupick** command (discussed later in the chapter) to retrieve the file.

For the following discussions, assume this information:

wombat	your login name
sys1	your system name
marie	recipients login name
sys2	recipients system name
money	file to be sent

Also assume that the two systems can communicate with each other.

To send the file **money** to login **marie** on system **sys2**, enter the following:

```
$ uuto money sys2!marie<CR>
$
```

The prompt on the second line is a signal that the file has been sent to a job queue. The job is now out of your hands; all you can do is wait for confirmation that the job reached its destination.

How do you know when the job has been sent? The easiest method is to alter the **uuto** command line by adding a **–m** option, as follows:

```
$ uuto –m money sys2!marie<CR>
$
```

This option sends a **mail** message back to you when the job has reached the recipients system. The message may look something like this:

```
$ mail<CR>
From uucp Thur Apr3 09:45 EST 1986
file /sys1/wombat/money, system sys1
copy succeeded
?
```

If you would like to check if the job has left your system, you can use the **uustat** command. This command keeps track of all the **uucp** and **uuto** jobs you submit and reports the status of each on demand. For example:

```
$ uustat<CR>
1145 wombat sys2 10/05-09:31 10/05-09:33 JOB IS QUEUED
$
```

The elements of this sample status message are as follows:

- **1145** is the job number assigned to the job of sending the file **money** to **marie** on **sys2**.
- **wombat** is the login name of the person requesting the job.
- **sys2** is the recipients system.
- **10/05-09:31** is the date and time the job was queued.
- **10/05-09:33** is the date and time this **uustat** message was sent.
- The final part is a status report on the job. Here, the report shows that the job has been queued, but has not yet been sent.

To receive a status report on only one **uuto** job, use the **–j** option and specify the job number on the command line:

uustat –j*jobnumber*<**CR**>

For example, to get a report on the job described in the previous example, specify 1145 (the job number) after the **–j** option:

```
$ uustat –j1145<CR>
1145 wombat sys2 10/05-09:31 10/05-09:37
COPY FINISHED,JOB DELETED
$
```

This status report shows that the job was sent and deleted from the job queue; it is now in the public directory of the recipients system. Other status messages and options for the **uustat** command are described in the *User's Reference Manual*.

That is all there is to sending files. To practice, try sending a file to yourself.

Tables 8-6 and 8-7 summarize the syntax and capabilities of the **uuto** and **uustat** commands, respectively.

Table 8-6. Summary of the **uuto** Command

Command Recap **uuto** – sends files to another login		
Command	**Options**	**Arguments**
uuto	**–m** and others*	*file system*!**login**
Description:	**uuto** sends a specified file to the public directory of a specified system, and notifies the intended recipient (by mail addressed to his or her login) that the file has arrived there.	
Remarks:	Files to be sent must have read permission for others; the files parent directory must have read and execute permissions for others. The **–m** option notifies the sender by mail when the file has arrived at its destination.	

* See the **uuto**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Table 8-7. Summary of the **uustat** Command

Command Recap **uustat** – checks job status of a **uucp** or **uuto** job		
Command	**Options**	**Arguments**
uustat	**–j** and others*	none
Description:	**uustat** reports the status of all **uucp** and **uuto** jobs you have requested.	
Remarks:	The **–j** option, followed by a job number, allows you to request a status report on only the specified job.	

* See the **uustat**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Receiving Files Sent with uuto: uupick Command

When a file sent by **uuto** reaches the public directory on your operating system, you receive a **mail** message. To continue the previous example, the owner of login **marie** receives the following mail message when the file **money** has arrived in her systems public directory:

```
$ mail
From uucp Wed May 14 09:22 EST 1986
/usr/spool/uucppublic/receive/marie/sys1//money
from sys1!wombat arrived
$
```

The message contains the following pieces of information:

- The first line tells you when the file arrived at its destination.
- The second line, up to the two slashes (*//*), gives the path name to the part of the public directory where the file has been stored.
- The rest of the line (after the two slashes) gives the name of the file and the sender.

Once you have disposed of the **mail** message, you can use the **uupick** command to store the file where you want it. Type the following command after the system prompt:

uupick<CR>

The command searches the public directory for any files sent to you. If it finds any, it reports the filename(s). It then prints a ? prompt as a request for further instructions from you.

For example, the owner of login **marie** issues the **uupick** command to retrieve the **money** file. The command responds:

```
$ uupick<CR>
from system sys1: file money
?
```

There are several available responses; we will look at the most common responses and what they do.

The first thing you should do is move the file from the public directory and place it in your login directory. To do so, type an **m** after the question mark:

```
?
m<CR>
$
```

This response moves the file into your current directory. If you want to put it in some other directory instead, follow the **m** response with the directory name:

```
?
m other_directory<CR>
```

If there are other files waiting to be moved, the next one displays, followed by the question mark. If not, **uucpick** returns a prompt.

If you do not want to do anything to that file now, press the RETURN key after the question mark:

```
?
<CR>
```

The current file remains in the public directory until the next time you use the **uupick** command. If there are no more messages, the system returns a prompt.

If you already know that you do not want to save the file, you can delete it by typing **d** after the question mark:

```
?
d<CR>
```

This response deletes the current file from the public directory and displays the next message (if there is one). If there are no additional messages about waiting files, the system returns a prompt.

Finally, to stop the **uupick** command, type a **q** after the question mark:

```
?
q<CR>
```

Any unmoved or undeleted files will wait in the public directory until the next time you use the **uupick** command.

Other available responses are listed in the *User's Reference Manual.*

Table 8-8 summarizes the syntax and capabilities of the **uupick** command.

Table 8-8. Summary of the **uupick** Command

Command Recap **uupick** – searches for files sent by **uuto** or **uucp**		
Command	**Options**	**Arguments**
uupick	available*	system name
Description:	**uupick** searches the public directory of your system for files sent by **uuto** or **uucp**. If any are found, the command displays information about the file and prompts you for a response.	
Remarks:	The question mark (**?**) at the end of the message shows that a response is expected. A complete list of responses is given in the *User's Reference Manual.*	

* See the **uupick**(1) manaul page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Networking

Networking is the process of linking computers and terminals so that users may be able to:

- log in on a remote computer as well as a local one
- log in and work on two computers in one work session (without alternately logging off one and logging in on the other)
- exchange data between computers

The commands presented in this section make it possible for you to perform these tasks. The **ct** command allows you to connect your computer to a remote terminal that is equipped with a modem. The **cu** command enables you to connect your computer to a remote computer, and the **uux** command lets you run commands on a remote system, without being logged in on it.

NOTE

On some small computers, the presence of these commands may depend on whether or not networking software is installed. If it is not installed on your system, you will receive a message like the following when you type a networking command:

```
cu: not found
```

Check with your System Administrator to verify the availability of networking commands on your operating system.

Connecting a Remote Terminal: ct Command

The **ct** command connects your computer to a remote terminal equipped with a modem, and allows a user on that terminal to log in. To do this, the command dials the phone number of the modem. The modem must be able to answer the call automatically. When **ct** detects that the call has been answered, it issues a **getty** (login) process for the remote terminal and allows a user on it to log in on the computer.

This command can be useful when issued from the opposite end, i.e., from the remote terminal itself. If you are using a remote terminal that is far from your computer and want to avoid long distance charges, you can use **ct** to have the computer place a call to your terminal. Simply call the computer, log in, and issue the **ct** command. The computer hangs up the current line and call your (remote) terminal back.

If **ct** cannot find an available dialer, it tells you that all dialers are busy and asks if it should wait until one becomes available. If you answer yes, it asks how long (in minutes) it should wait for one.

Command Line Format

To execute the **ct** command, use the followiwng format:

ct [*options*] *telno*<**CR**>

The argument *telno* is the telephone number of the remote terminal.

Sample Command Usage

You are logged in on a computer through a local terminal and you want to connect a remote terminal to your computer. The phone number of the modem on the remote terminal is 932-3497. Enter this command line:

ct –h –w5 –s1200 9=9323497<CR>

NOTE

The equal sign (=) represents a secondary dial tone, and dashes (–) following the phone number represent delays (the dashes are useful following a long distance number).

ct calls the modem, using a dialer operating at a speed of 1200 baud. If a dialer is not available, the **–w5** option causes **ct** to wait for a dialer for five minutes before quitting. The **–h** option tells **ct** not to disconnect the local terminal (the terminal on which the command was issued) from the computer.

Now you want to log in on the computer from home. To avoid long distance charges, use **ct** to have the computer call your terminal:

ct –s1200 9=9323497<CR>

Because you did not specify the **–w** option, if no device is available, **ct** sends you the following message:

```
1 busy dialer at 1200 baud Wait for dialer?
```

If you type **n** (no), the **ct** command exits. If you type **y** (yes), **ct** prompts you to specify how long **ct** should wait:

```
Time, in minutes?
```

If a dialer is available, **ct** responds with:

```
Allocated dialer at 1200 baud
```

This means that a dialer has been found. In any case, **ct** asks if you want the line connecting your remote terminal to the computer to be dropped:

```
Confirm hangup?
```

If you type **y** (yes), you are logged off and **ct** calls your remote terminal back when a dialer is available. If you type **n** (no), the **ct** command exits, leaving you logged in on the computer.

Table 8-9 summarizes the syntax and capabilities of the **ct** command.

Table 8-9. Summary of the **ct** Command

Command Recap **ct** – connect computer to remote terminal		
Command	**Options**	**Arguments**
ct	**–h, –w, –s** and others*	*telno*
Description:	**ct** connects the computer to a remote terminal and allows a user to log in from that terminal.	
Remarks:	The remote terminal must have a modem capable of answering phone calls automatically.	

* See the **ct**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Calling Another Operating System: cu Command

The **cu** command connects a remote computer to your computer and allows you to be logged in on both computers simultaneously. This means that you can move back and forth between the two computers, transferring files and executing commands on both, without dropping the connection.

The method used by the **cu** command depends on the information you specify. You must specify the telephone number or system name of the remote computer. If you specify a phone number, it is passed on to the automatic dial modem. If you specify a system name, **cu** obtains the phone number from the **Systems** file. If an automatic dial modem is not used to establish the connection, the line (port) associated with the direct link to the remote computer can be specified on the command line.

Once the connection is made, the remote computer prompts you to log in on it. When you have finished working on the remote terminal, log off it and terminate the connection by typing <~.>. You are still logged in on the local computer.

NOTE

The **cu** command is not capable of detecting or correcting errors; data may be lost or corrupted during file transfers. After a transfer, you can check for loss of data by running the **sum** command or the **ls –l** command on the file that was sent and the file that was received. Both of these commands report the total number of bytes in each file; if the totals match, your transfer was successful. The **sum** command checks more quickly and gives output that is easier to interpret. (See the **sum**(1) and the **ls**(1) manual pages in the *User's Reference Manual* for details.)

Command Line Format

To execute the **cu** command, use the following format:

cu [*options*] *telno* | *systemname*<**CR**>

where:

telno
is the telephone number of a remote computer.

Equal signs (=) represent secondary dial tones; dashes (–) represent four-second delays.

systemname
is a system name that is listed in the **Systems** file.

The **cu** command obtains the telephone number and baud rate from the **Systems** file and searches for a dialer. The **–s**, **–n**, and **–l** options should not be used with *systemname*. (To see the list of computers in the **Systems** file, run the **uuname** command.)

Once your terminal is connected and you are logged in on the remote computer, all standard input (input from the keyboard) is sent to the remote computer. Table 8-10 shows the commands you can execute while connected to a remote computer through **cu**.

Table 8-10. Command Strings for Use with **cu**

String	Interpretation
~.	Terminate the link.
~!	Escape to the local computer without dropping the link. To return to the remote computer, type <**^d**> (**CTRL-d**).
~!*command*	Execute *command* on the local computer.
~**$***command*	Run *command* locally and send its output to the remote system.
~**%cd** *path*	Change the directory on the local computer where *path* is the path name or directory name.
~**%take** *from* [*to*]	Copy a file named *from* (on the remote computer) to a file named *to* (on the local computer). If *to* is omitted, the *from* argument is used in both places.
~**%put** *from* [*to*]	Copy a file named *from* (on the local computer) to a file named *to* (on the remote computer). If *to* is omitted, the *from* argument is used in both places.
~~...	Send a line beginning with ~ (~~...) to the remote computer.
~**%break**	Transmit a **BREAK** to the remote computer (can also be specified as ~**%b**).

Table 8-10. Command Strings for Use with **cu** (cont'd)

String	Interpretation
~%nostop	Turn off the handshaking protocol for the remainder of the session. This is useful when the remote computer does not respond properly to the protocol characters.
~%debug	Turn the **–d** debugging option on or off (can also be specified as **~%d**).
~t	Display the values of the terminal I/O (input/output) structure variables for your terminal (useful for debugging).
~l	Display the values of the termio structure variables for the remote communication line (useful for debugging).

NOTE

The use of **~%put** requires **stty** and **cat** on the remote computer. It also requires that the current erase and kill characters on the remote computer are identical to the current ones on the local computer.

The use of **~%take** requires the existence of the **echo** and **cat** commands on the remote computer. Also, **stty tabs** mode should be set on the remote computer if tabs are to be copied without expansion.

Sample Command Usage

You want to connect your computer to a remote computer called **eagle**. The phone number for eagle is 555–7867. Enter the following command line:

cu –s1200 9=5557867<CR>

The **–s1200** option causes **cu** to use a 1200 baud dialer to call **eagle**. If the **–s** option is not specified, **cu** uses a dialer at the default speed, 300 baud.

When **eagle** answers the call, **cu** notifies you that the connection has been made, and prompts you for a login ID:

```
connected
login:
```

Enter your login ID and password.

The **take** command allows you to copy files from the remote computer to the local computer. For example, you want to make a copy of a file named **proposal** for your local computer. The following command copies **proposal** from your current directory on the remote computer and places it in your current directory on the local computer. If you do not specify a file name for the new file, it is also called **proposal**.

~%take proposal<CR>

The **put** command allows you to do the opposite: copy files from the local computer to the remote computer. For example, you want to copy a file named **minutes** from your current directory on the local computer to the remote computer, type:

~%put minutes minutes.9–18<CR>

Here, you specified a different name for the new file (**minutes.9-18**). Therefore, the copy of the **minutes** file that is made on the remote computer is called **minutes.9-18**.

Table 8-11 summarizes the syntax and capabilities of the **cu** command.

Table 8-11. Summary of the **cu** Command

Command Recap **cu** – connects computer to remote computer		
Command	**Options**	**Arguments**
cu	**–s** and others*	*telno* (or) *systemname*
Description:	**cu** connects your local computer to a remote computer and allows you to be logged in on both simultaneously. Once you are logged in, you can move between computers to execute commands and transfer files on each without dropping the link.	

* See the **cu**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Executing on a Remote System: uux Command

The command **uux** allows you to execute operating system commands on remote computers. It can gather files from various computers, execute a command on a specified computer, and send the standard output to a file on a specified computer. The execution of certain commands may be restricted on the remote machine. The command notifies you by mail if the command you have requested is not allowed to execute.

Command Line Format

To execute the **uux** command, use the following format:

uux [*options*] *command–string*<**CR**>

The *command–string* is made up of one or more arguments. All special shell characters (e.g., "<>|^") must be quoted, either by quoting the entire *command–string* or quoting the character as a separate argument. Within the *command–string*, the command and file names may contain a *system name!* prefix.

All arguments that do not contain a *systemname* are interpreted as command arguments. A file name may be either a full path name or the name of a file under the current directory (on the local computer).

Sample Command Usage

If your computer is hard-wired to a larger host computer, you can use **uux** to get printouts of files that reside on your computers by entering:

pr minutes | uux –p host!lp<**CR**>

This command line queues the file **minutes** to be printed on the area printer of the computer **host**.

Table 8-12 summarizes the syntax and capabilities of the **uux** command.

Table 8-12. Summary of the **uux** Command

Command Recap **uux** – executes commands on a remote computer		
Command	**Options**	**Arguments**
uux	**–1, –p**, and others*	*command-string*
Description:	**uux** allows you to run operating system commands on remote computers. It can gather files from various computers, run a command on a specified computer, and send the standard output to a file on a specified computer.	
Remarks:	By default, users of the **uux** command have permission to run only the **mail** and **mailx** commands. Check with your System Administrator to find out if users on your system have been granted permission to run other commands.	

* See the **uux**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

9 ksh Tutorial

Introduction

ksh is a direct descendant of the Form shell with most of the form entry/edit features removed and with many features added. The primary focus is to provide an enhanced programming environment in addition to the major command entry features of **Csh**. Many of the additions are provided so that medium sized programming tasks can be written at the shell level without a serious performance penalty. A concerted effort has been made to achieve SYSTEM V/88 shell compatibility so that scripts written for the SYSTEM V/88 shell can run without modification with **ksh**.

Shell Variables

The ability to define and use variables to store and retrieve values is an important feature in most programming languages. **ksh** has variables with *identifier* names that follow the same rules as the SYSTEM V/88 shell. Since all variables have string representations, there is no need to specify the *type* of each variable in the shell. In **ksh**, each variable can have one or more *attributes* that control the internal representation of the variable, the way the variable is printed, and its access or scope. Two of the attributes, *readonly* and *export*, are available in the SYSTEM V/88 shell. The **typeset** built-in command of **ksh** assigns attributes to variables. The complete list of attributes, some of which are discussed here, appears in the manual page. The **unset** built-in of the **ksh** removes values and attributes of parameters.

Whenever a value is assigned to a variable, the value is transformed according to the attributes of the variable. Changing the attribute of a variable can change its value. There are three attributes for field justification, as might be needed for formatting a report. For each of these attributes, a width can be defined explicitly or it is defined the first time an assignment is made to the variable. Each assignment causes justification of the field, truncating if necessary. Assignment to fixed sized variables provides a simple way to generate a substring consisting of a fixed number of characters from the beginning or end of a string.

The attributes **–u** and **–l**, are used for uppercase and lowercase formatting, respectively. Since it makes no sense to have both attributes on simultaneously, turning on either of these attributes turns the other off. The following script provides an example of the use of shell variables with attributes. This script reads a file of lines each consisting of five fields separated by : and prints fields 4 and 2 in uppercase in columns 1-15, left justified, and columns 20-25 right-justified respectively.

```
typeset -L15u f4                    # 15 character left justified
typeset -R6u f2                     # 6 character right justified
IFS=:
set -f                              # skip file name generation
while  read -r f1 f2 f3 f4 f5       # read line, split into fields
do      print -r "$f4  $f2"         # print fields 4 and 2
done
```

The integer attribute, **–i**, causes the variable to be internally represented as an integer. The **i** can be followed by a number representing the numeric base for printing, otherwise, the first assignment to an *integer* variable defines the output *base* (see below). This base is used whenever the variable is printed. Assignment to *integer* typed variables result in arithmetic evaluation, as described below, of the right hand side.

ksh allows one-dimensional *arrays* in addition to simple variables. Any variable can become an array by referring to it with a *subscript*. All elements of an array need not exist. Subscripts for arrays must evaluate to an integer between 0 and 511, otherwise, an error results. Evaluation of subscripts is described in the next section. Attributes apply to the whole array.

Assignments to array variables can be made with parameter assignment statements or with the **typeset** built-in. Referencing of subscripted variables requires the character **$**, but also requires braces around the array element name. The braces are needed to avoid conflicts with the file name generation mechanism. The form of any array element reference is:

${*name* **[** *subscript* **]}**

A subscript value of * or @ can be used to generate all elements of an array, as they are used for expansion of positional parameters.

A few additional operations are available on shell variables. **${#***name***}** will be the length in bytes of **$***name*. For an array variable **${#***name***[*]}** gives the number of elements in the array.

There are four parameter substitution modifiers that have been added to strip off leading and trailing substrings during parameter substitution. The modifier **#(##)** strips off the smallest (largest) matching pattern from the left and the modifier **%(%%)** strips off the smallest (largest) matching pattern from the right. For example, if the shell variable **i** has value **file.c**, the expression **${i%.c}.o** has value **file.o**.

Arithmetic Evaluation

The built-in command, **let**, provides the ability to do integer arithmetic. All arithmetic evaluations are performed using *long* arithmetic. Arithmetic constants are written as

base#number

where *base* is a decimal integer between 2 and 36 and *number* is any non-negative number. Anything after a decimal point is truncated. Base 10 is used if no base is specified.

Arithmetic expressions are made from constants, variables, and one or more of the 14 operators listed in the manual page. Operators are evaluated in order of precedence. Parentheses may be used for grouping. A variable does not have to have an integer attribute to be used within an arithmetic expression. The name of the variable is replaced by its value within an arithmetic expression. The following statement can be used to increment a variable **x**:

```
let x=x+1
```

Note that there is no space before or after the operators + and =. This is because each argument to **let** is an expression to evaluate. The last expression determines the value returned by **let**. Let returns true if the last expression evaluates to a non-zero value. Otherwise, **let** returns false.

Many of the arithmetic operators have special meaning to the shell and must be quoted. Since this can be burdensome, an alternate form of arithmetic evaluation syntax has been provided. For any command that begins with **((** , all characters until the matching **))** are treated as a quoted arithmetic expression. The double parentheses usually avoids incompatibility with the SYSTEM V/88 shells use of parentheses for grouping a set of commands to be run in a sub-shell.

Expressions inside double parentheses can contain blanks and special characters without quoting:

((...))

is equivalent to:

let " ... "

The following script prints the first *n* lines of its standard input onto its standard output, where *n* can be supplied as an optional argument whose default value is 20:

```
typeset -i n=${1-20}                          # set n
while   read -r line && (( (n=n-1)>=0 ))      # at most n lines
do      print -r - "$line"
done
```

Functions and Command Aliasing

Two mechanisms are provided for creating pseudo-commands, i.e., things that look like commands, but do not always create a process. The first technique is called command name *aliasing*.

As a command is being read, the command name is checked against a list of *alias* names. If it is found, the name is replaced by the text associated with the *alias* and then rescanned. The text of an alias is not checked for aliases so recursive definitions are not allowed. However, if the value of an alias ends in a space, the word following the alias is also checked for alias substitution.

Aliases are defined with the **alias** built-in. The form of an **alias** command definition is:

alias *name* = *value*

The first character of an *alias* name can be any non-special printable character, while all remaining characters must be alpha-numeric. The replacement text, *value*, can contain any valid shell script, including meta-characters such as pipe symbols and i/o-redirection. Unlike **csh**, aliases in **ksh** cannot take arguments. Aliases can be used to redefine built-in commands so that the alias can be used to look for *test* in your current working directory instead of using the built-in **test** command.

alias test=./test

Keywords such as **for** and **while** cannot be changed by aliasing. The command **alias**, without arguments, generates a list of aliases and corresponding texts. The **unalias** command removes the name and text of an alias.

Aliases are used to save typing and to improve readability of scripts. For example, the alias **alias integer='typeset –i'** allows integer the variables **i** and **j** to be declared and initialized with the command **integer i=0 j=1** .

Aliases can be used to bind program names to the full path name of the program. This eliminates the path search but requires knowledge of where that program will be stored. *Tracked* aliases make this use for aliasing automatic. A tracked alias is not given a value. Its value is defined at the first reference by a path-search as the full path name equivalent of the name, and remains defined until the PATH variable is changed. Programs found in directories that do not begin with **/** that occur earlier in the path-search than the value of the tracked alias, take precedence over tracked aliases.

Tracked aliases provide an alternative to the **Csh** command hashing facility. Tracked aliases do not require time for initialization and allow for new commands to be introduced without the need for rehashing. The **–h** option to the shell allows all command names that are valid alias names to become tracked aliases. This option is automatically turned on for non-interactive shells.

Functions are more general than aliases but also more costly. Functions definitions are of the form:

function *name*
{
any shell script
}

The function is invoked by writing *name* and optionally following it with arguments. Positional parameters are saved before each function call and restored when completed. Functions are executed in the current shell environment and can share named variables with the calling program. Options, other than execution trace **–x**, set by the calling program are passed down to a function. The option flags are not shared with the function so that any options set within a function are restored when the function exits. All traps other than EXIT and ERR (described later) are also inherited. A trap on EXIT within a function executes after the function completes, but before the caller resumes. Therefore, any variable assignments and any options set as part of a trap action are effective after the caller resumes. The **return** built-in can be used to cause the function to return to the statement following the point of invocation.

By default, variables are inherited by the function and shared by the calling program. However, environment substitutions preceding the function call apply only to the scope of the function call. Also, variables defined with the **typeset**, built-in command are local to the function that they are declared in. The following function defined is invoked as **y=13 name**, **x** and **y** are local variables with respect to the function **name**, and **z** is global:

```
function  name
{
     typeset -i x=10
     let z=x+y
     print $z
}
```

Alias and function names are never directly carried across separate invocations of **ksh**, but can be passed down to sub-shells. Ordinarily, shell scripts invoked by name are executed in a sub-shell, while scripts invoked as **ksh** *script* and shell escapes from other programs are carried out by a separate shell invocation. The **–x** flag is used with **alias** to carry aliases to sub-shells, while the **–fx** flags of **typeset** are used to do the same for functions.

Each user can create a startup file for aliases and functions or any other commands. Aliases and functions that are to be available for all shell invocations should be put into this file. Aliases and functions that should apply to scripts as well as interactive use, should be set with the **–x** flag. Setting this flag to redefine the semantics of a command can have undesired side effects. For example, **alias –x ls='ls –l'** will cause shell procedures that use the **ls** command within a pipeline to break. By setting and exporting the environment variable, ENV, to the name of this file, the aliases and functions are defined each time **ksh** is invoked. The value of the ENV variable undergoes parameter substitution before to its use.

Several of the operating system commands can be aliased to **ksh** built-ins. Some of these are automatically set each time the shell is invoked. In addition, about 20 frequently used operating system commands are set as tracked aliases.

The location of an alias command can be important since aliases are only processed when a command is read. A procedure is read all at once (unlike *profiles* that are read a command at a time) so that any aliases defined there do not effect any commands within this script.

A name is checked to see if it is a built-in command before checking to see if it is a function. To write a function to replace a built-in command you must define a function with a different name and alias the built-in name to this function. For example, to write a **cd** function that changes the directory and prints out the directory name, you can write:

```
alias cd=_cd
function  _cd
{
     if       'cd'  "$@"
     then     echo  $PWD
     fi
}
```

The single quotes around **cd** within the function prevents alias substitution. The PWD variable is described below.

The combination of aliases and functions can be used to do things that cannot be done with either of these separately. For example, the following defined function and aliases allow you to write loops:

```
function _from # i=start to finish [ by incr]
{
        typeset var=${1%%=*}
       integer incr=${5-1} $1
       while   (( $var <= $3 ))
       do      _repeat
               let $var=$var+incr
       done
}
alias repeat='function _repeat {'  from='}; _from'
```

The following example shows the expected behavior:

```
repeat
        any script command
from i=1 to 13 by 3
```

Input and Output

An extended I/O capability enhances the use of the shell as a programming language. The operating system shell has a built-in **read** for reading lines from file descriptor 0, but does not have any internal output mechanism. As a result, the **echo**(1) command is used to produce output for a shell procedure; this is inefficient and also restrictive. For example, there is no way to read in a line from a terminal and *echo* the line exactly as is. In the operating system shell, the **read** built-in cannot be used to read lines that end in \ ; the **echo** command treats certain sequences as control sequences. In addition, there is no way to have more than one file open at any time for reading.

ksh has options on the **read** command to specify the file descriptor for the input. The **exec** built-in can be used to open, close, and duplicate file streams. The **–r** option allows a \ at the end of an input line to be treated as a regular character instead of the line continuation character. The first argument of the **read** command can be followed by a **?** and a prompt to produce a prompt at the terminal before the read. If the input is not from a terminal device, the prompt is not issued.

The **ksh** built-in, **print**, is used to output characters to the terminal or to a file. Again, it is possible to specify the file descriptor number as an option to the command. Usually, the arguments to this command are processed the same as for **echo**(1). However, the **–r** flag can be used to output the arguments without any special meaning. The **–n** flag can be used here to suppress the trailing new-line that is usually appended.

To improve performance of existing shell programs, the **echo** command is built into **ksh**. For the SYSTEM V/88 version of **ksh**, the built-in **echo** is equivalent to:

print –

where the **–** signifies that there are no more options permitted. On the Berkeley UNIX version, the value of the PATH variable determines the behavior of the built-in **echo** command. If **echo** would resolve to **/bin/echo** with a path search, then **echo** is equivalent to:

print –R.

The **–R** option allows only the **–n** flag to be recognized as the next argument. Otherwise, **echo** behaves like the SYSTEM V/88 **echo** command.

The shell is frequently used as a programming language for interactive dialogues. The **select** statement makes it easier to present menu selection alternatives to the user and evaluate the reply. The list of alternatives is numbered and put in columns. A user settable prompt, **PS3**, is issued and if the answer is a number corresponding to one of the alternatives, the select loop variable is set to this value. In any case, the REPLY variable is used to store the user entered reply. The shell variables LINES and COLUMNS are used to control the layout of select lists.

Command Re-entry

An interactive shell saves the commands you type at a terminal in a file. If the variable HISTFILE is set to the name of a file to which the user has write access, the commands are stored in this *history* file. Otherwise, the file **$HOME/.sh_history** is checked for write access; if this fails, an unnamed file is used to hold the history lines. This file may be truncated if this is a top level shell.

The number of commands accessible to the user, is determined by the value of the HISTSIZE variable at the time the shell is invoked. The default value is 128. A command may consist of one or more lines since a compound command is considered one command. If the character **!** is placed within the *primary prompt* string, **PS1**, it is replaced by the command number each time the prompt is given. Whenever the history file is named, all shells which use this file share access to the same history.

A built-in command **fc** (fix command) is used to list and/or edit any of these saved commands. The command can always be specified with a range of one or more commands. The range can be specified by giving the command number, relative or absolute, or by giving the first character or characters of the command. The option **–l** is used to specify listing of previous commands. When given without specifying the range, the last 16 commands are listed, each preceded by the command number.

If the listing option is not selected, the range of commands specified, or the last command if no range is given, is passed to an editor program before being re-executed by **ksh**. The editor to be used may be specified with the option **/-e**, following it with the editor name. If this option is not specified, the value of the shell variable FCEDIT is used as the name of the editor, providing this variable has non-null value. If this variable is not set or is null and the **–e** option has not been selected, then **/bin/ed** is used. When editing has been complete, the edited text automatically becomes the input for **ksh**. As this text is read by **ksh**, it is echoed onto the terminal.

An editor name of **–** is used to bypass the editing and re-execute the command. Here, only a single command can be specified as the range and an optional argument of the form *old=new* may be added that requests a simple string substitution prior to evaluation. The following alias that has been pre-defined so that the single key-stroke **r** can be used to re-execute the previous command.

alias r='fc –e –'

The key-stroke sequence, **r abc=def c**, can be used to re-execute the last command that starts with the letter **c** with the first occurrence of the string **abc** replaced with the string **def**. Typing **r c > file** re-executes the most recent command starting with the letter **c**, with standard output redirected to *file*.

In-line Editing

Lines typed from a terminal frequently need changes made before entering them. With the SYSTEM V/88 shell, the only method to fix up commands is by backspacing or killing the whole line. **ksh** offers options that allow the user to edit parts of the current command line before submitting the command. The in-line edit options make the command line into a single line screen edit window. When the command is longer than the width of the terminal, only a portion of the command is visible. Moving within the line automatically makes that portion visible. Editing can be performed on this window until the RETURN key is pressed. The editing modes have commands that access the history file in which previous commands are saved. A user can copy any of the most recent HISTSIZE commands from this file into the input edit window. You can locate commands by searching or by position.

The in-line editing options do not use the *termcap* database. They work on most standard terminals. They only require that the BACKSPACE character moves the cursor left and the SPACE character overwrites the current character on the screen and moves the cursor to the right.

There is a choice of editor options. The *emacs*, *gmacs*, or *vi* option is selected by turning on the corresponding option of the **set** command. If the value of the EDITOR or VISUAL ends any of these suffixes, the corresponding options is turned on. A large subset of each of each of these editors features are available within the shell. Additional functions, e.g., file name completion, are also available.

In the *emacs* or *gmacs* mode, the user positions the cursor to the point needing correction and inserts, deletes, or replaces characters as needed. The only difference between these two modes is the meaning of the command **^T**. CONTROL keys and escape sequences are used for cursor positioning and control functions. The available editing functions are listed in the manual page.

The *vi* editing mode starts in insert mode and enters control mode when the user types **ESC (033)**. The RETURN key, which submits the current command for processing, can be entered from either mode. The cursor can be anywhere on the line. A subset of commonly used *vi* commands are available. The **k** and **j** command that normally move up and down by one *line*, move up and down one *command* in the history file, copying the command into the input edit window. For reasons of efficiency, the terminal is kept in canonical mode until an **ESC** is typed. On some terminals, and on

earlier versions of the SYSTEM V/88 operating system, this does not work correctly. The **viraw** option of the **set** command, which always uses **raw** or **cbreak** mode, must be used in this case.

Most of the code for the editing options does not rely on the **ksh** code and can be used in a stand-alone mode with most any command to add in-line edit capability. However, all versions of the in-line editors have some features that use some shell specific code. For example, **ESC-=** in all edit modes prints the names of files that match the current word; **ESC-*** adds the expanded list of matching files to the command line. A trailing * is added to the word if it does not contain any file pattern matching characters before the expansion.

Job Control

The job control feature allows the user to stop and restart programs, and to move programs to and from the foreground and the background. It only works on systems that provide support for these features. However, even systems without job control have a **monitor** option which, when enabled, reports the progress of background jobs and enables the user to **kill** jobs by job number or job name.

An interactive shell associates a *job* with each pipeline typed in from the terminal and assigns them a small integer number called the job number. If the job is run asynchronously, the job number is printed at the terminal. At any given time, only one job owns the terminal, i.e., keyboard signals are only sent to the processes in one job. When **ksh** creates a foreground job, it gives it ownership of the terminal. If you are running a job and want to stop it, press the key ^**Z** (CTRL-**Z**) which sends a STOP signal to all processes in the current job. The shell receives notification that the processes have stopped and takes back control of the terminal.

There are commands to continue programs in the foreground and background and several ways to refer to jobs. The character **%** introduces a job name. You can refer to jobs by name or number as described in the manual page. The built-in command **bg** allows you to continue a job in the background, while the built-in command **fg** allows you to continue a job in the foreground even though you may have started it in the background.

A job being run in the background stops if it tries to read from the terminal. It is also possible to stop background jobs that try to write on the terminal by setting the terminal options appropriately.

There is a built-in command **jobs** that lists the status of all running and stopped jobs. In addition, you are notified of the change of state of any background jobs just before each prompt. When you try to leave the shell while jobs are stopped or running, you receive a message from **ksh**. If you ignore this message and try to leave again, all stopped processes are terminated.

A built-in version of **kill** makes it possible to use *job* numbers as targets for signals. Signals can be selected by number or name. The name of the signal is the name found in the *include* file **/usr/include/signal.h** with the prefix **SIG** removed. The **–l** flag of **kill** generates a list of valid signal numbers and names.

Security

There are several documented problems associated with the security of shell procedures. These security holes occur primarily because a user can manipulate the *environment* to subvert the intent of a *setuid* shell procedure. Frequently, shell procedures are initiated from binary programs, without the author's awareness, by library routines that invoke shells to carry out their tasks. When the binary program is run *setuid*, the shell procedure runs with the permissions afforded to the owner of the binary file.

In the SYSTEM V/88 shell, the **IFS** parameter is used to split each word into separate command arguments. If a user knows that some *setuid* program will run **sh –c /bin/pwd** (or any other command in **/bin**), then the user sets and exports **IFS=/**. Instead of running **/bin/pwd**, the shell runs **bin** with **pwd** as an argument. The user puts their own **bin** program into the current directory. This program can create a copy of the shell, make this shell *setuid*, then run the **/bin/pwd** program so that the original program continues to run successfully. This kind of penetration is not possible with **ksh-i** since the **IFS** parameter only splits arguments that result from command or parameter substitution.

Some *setuid* programs run programs using **system**() without giving the full path name. If the user sets the PATH variable so that the desired command is found in their local **bin**, the same technique described above can be employed to compromise the security of the system. To close up this and other security holes, **ksh** goes into a *protected* mode whenever the real and effective user or

group id are not the same. In this mode, the PATH variable is reset to a default value and the **.profile** and **ENV** files are not processed. Instead, the file **/etc/suid_profile** is read and executed. This gives an administrator control over the environment to set the PATH variable or to log setuid shell invocations. Clearly, security of the system is compromised if **/etc** or this file is publicly writable.

In BSD UNIX, the operating system looks for the characters **#!** as the first two characters of an executable file. If these characters are found, the next word on this line is taken as the interpreter to **exec** for this command and the interpreter is **exec**ed with the name of the script as argument zero and argument one. If the *setuid* or *setgid* bits are on for this file, then the interpreter is run with the effective uid and/or gid set accordingly. This scheme has two major drawbacks. First, using the **#!** notation forces an **exec** of the interpreter even when the call is invoked from the interpreter which it must execute. This is inefficient because the interpreter can handle a failed **exec** much faster than starting up again. More importantly, *setuid* and *setgid* procedures provide an easy target for intrusion. By linking a *setuid* or *setgid* procedure to a name beginning with a **–**, the interpreter is fooled into thinking that is being invoked with a command line option instead of the name of a file. When the interpreter is the shell, the user gets a privileged interactive shell. There is code in **ksh** to guard against this simple form of intrusion.

A more reliable way to handle *setuid* and *setgid* procedures is provided with **ksh**. The technique does not require any changes to the operating system and provides better security. Another advantage to this method is that it also allows scripts that have execute permission but no read permission to run. Taking away read permission makes scripts more secure.

The method relies on a **setuid** root program to authenticate the request and execute the shell with the correct mode bits to carry out the task. This shell is invoked with the requested file already open for reading. A script that cannot be opened for reading or has its **suid** and/or **setgid** bits turned on, causes this **setuid** root program to get executed. For security reasons, this program is given the full pathname **/etc/suid_exec**.

Miscellaneous

ksh has several additional features to enhance functionality and performance. This section lists most of these features.

Tilde Substitution

The character ~ at the beginning of a word has special meaning to **ksh**. If the characters after the ~ up to a **/** match a user login name in the **/etc/passwd** file, the ~ and the name are replaced by that users login directory. If no match is found, the original word is unchanged. A ~ by itself, or in front of a **/**, is replaced by the value of the HOME parameter. A ~ followed by a + or **–** is replaced by the value of the parameter PWD and OLDPWD, respectively. Tilde substitution takes place when the script is read, not while it is executed.

Built-in I/O Redirection

All built-in commands can be redirected. Compound commands that are redirected are not carried out in a separate process.

Options

All options have names that can be used in place of flags for setting and resetting options. The command **set –o** lists the current option settings.

The option, **–f** or **noglob**, is used to disable file name generation.

The option **ignoreeof** can be used to prevent **^D** from exiting the shell and possibly logging you out. You must type **exit** to log out.

The **–h** or **trackall** option causes all commands whose name is a valid alias name to become a *tracked* alias. This option is automatically turned on for non-interactive shells.

The job **monitor** option causes a report to be printed before issuing the next prompt when each background job completes. It is automatically enabled for systems that have job control.

If the **bgnice** option is set, background jobs are run at a lower priority.

The option **markdirs** causes a trailing **/** to be appended on every directory name resulting from a pattern match.

The **protected** or **–p** options provides additional security by disabling the **ENV** from being executed and by resetting the PATH variable to the default value. Whenever a shell is run with the effective **uid** (gid) not equal to the real **uid** (gid), this option is implicitly enabled. Instead of the **ENV** file, the file **/etc/suid_profile** is read so that administrators can have control over **setuid** scripts.

Built-in pwd

The **pwd** command is built-into **ksh** and therefore, much faster.

Built-in fexpr

The **fexpr** command is a built-in version of **/bin/expr**, and therefore, much faster.

Built-in getopts

The **getopts** is a built-in version of **/usr/bin/getopt**. For details of the differences between the two commands, see **getopts**(1).

Logical Naming

The **cd** command takes you where you expect to go even if you cross symbolic links. Thus, **cd ..** moves you up one level closer to the root even if your current directory is a symbolic link.

Previous Directory

ksh remembers your last directory in the variable OLDPWD. The **cd** built-in can be given with argument **–** to return to the previous directory and prints the name of the directory. Note that **cd –** done twice, returns you to the starting directory, not the second previous directory. A directory *stack* manager written as shell *functions*, *push* and *pop* directories from the stack.

Additional Variables and Parameters

Several new parameters have special meaning to **ksh**. The variable PWD is used to hold the current working directory of the shell. The variable OLDPWD is used to hold the previous working directory of the shell.

The variable FCEDIT is used by the **fc** built-in described above. The variables VISUAL and EDITOR are used to determine the edit modes described above.

The variable ENV is used to define the startup file for non-login **ksh** invocations.

The variables HISTSIZE and HISTFILE control the size and location of the file containing commands entered at a terminal.

The parameter MAILPATH is a colon (:) separated list of file names to be checked for changes periodically. The user is notified before the next prompt. Each of the names in this list can be followed by a **?** and a prompt to be given when a change has been detected in the file. The prompt is evaluated for parameter substitution. The parameter **$_** within a mail message evaluates to the name of the file that has changed. The parameter MAILCHECK is used to specify the minimal interval in seconds before checking for new mail.

The variable RANDOM produces a random number each time it is referenced. Assignment to this variable sets the seed for the random number generator.

The variable SECONDS is incremented every second. In a roundabout way, this variable can be used to generate a time stamp into the **PS1** prompt. The following code explains how you can do this on SYSTEM V/88.

```
#If you . this script then you can use $TIME as part of your PS1 string
#to get the time of day in your prompt
typeset -RZ2  _x1 _x2 _x3
let SECONDS=$(date  '+3600*%H+60*%M+%S')
_s='(_x1=(SECONDS/3600)%24)==(_x2=(SECONDS/60)%60)==(_x3=SECONDS%60)'
TIME='"${_d[_s]}$_x1:$_x2:$_x3"'
# PS1=${TIME}whatever
```

The parameter PPID is used to generate the process id of the process that invoked this shell.

The value of the parameter _ is the last argument of the previous foreground command. Before executing each command, this parameter is set to the file name of the command and placed in the environment.

The parameter TMOUT can be set to be the number of seconds that the shell waits for input before terminating. A 60 second warning message is printed before terminating.

The COLUMNS variable can be used to adjust the width of the edit window for the in-line edit modes. It is also used by the **select** command to present menu choices.

The LINES variable controls how many rows a select list takes up on the screen. Select lists try to occupy no more then two-thirds of LINES lines on the screen.

Modified Variables

The input field separator parameter, IFS, is only used to split words that have undergone parameter or command substitution. In addition, adjacent non-blank delimiters separate null fields in **ksh**.

The PS1 parameter is evaluated for parameter substitution and a **!** is replaced by the current command number.

Timing Commands

A keyword **time** replaces the **time** command. Any function, command, or pipeline can be preceded by this keyword to obtain information about the elapsed, user and system times. Since I/O redirection bind to the command, not to **time**, parenthesis should be used to redirect the timing information that is normally printed on file descriptor 2.

Co-process

ksh can spawn a *co-process* by adding a **| &** after a command. This process is run with its standard input and standard output connected to the shell. The built-in command **print** with the **–p** option writes into the standard input of this process and the built-in command **read** with the **–p** option reads from the output of this process. Only one such process can exist at any time.

Process Substitution

This feature is only available on versions of the UNIX operating system that support the **/dev/fd** directory for naming open files. (This feature is not available on SYSTEM V/88.) Each command argument of the form **(***list***)**, **<(***list***)**, or **>(***list***)** runs process *list* asynchronously connected to some file in the **/dev/fd** directory. The name of this file becomes the argument to the command. If the form with > is selected, writing on this file provides input for *list*. If < is used or omitted, the file passed as an argument contains the output of the *list* process. For example, the following *cuts* fields 1 and 3 from the files *file1* and *file2* respectively, *pastes* the results together, and sends it to the processes *process1* and *process2*, as well as putting it onto the standard output.

paste (cut –f1 *file1***) (cut –f3** *file2***) | tee >(***process1***) >(***process2***)**

Note that the file that is passed as an argument to the command is a **pipe**(2) so that the programs that expect to **lseek**(2) on the file will not work.

Command Substitution

Command substitution (` `) in the SYSTEM V/88 shell has some complicated quoting rules. It is hard to write a **sed** pattern that contains back slashes within command substitution. Putting the pattern is single quotes does not help; **ksh** leaves the SYSTEM V/88 shell command substitution alone and adds a newer and easier to use command substitution syntax. All characters between a **$(** and a matching **)** are evaluated as a command the output is substituted just as with ` `. The **($** means *value of* and the **( )** denotes a command. The command itself can contain quoted strings even if the substitution occurs within double quotes; nesting is legal. You can use unbalanced parenthesis within the command providing they are quoted.

The special command substitution of the form **$(cat file)** can be replaced by **$(< file)**, which is faster because no separate process is created.

Whence

The *aliases*, *functions*, and built-ins makes it substantially more difficult to know what a given command word really means. A built-in command, **whence**, when used with the **–v** option answers this question. A line is printed for each argument to **whence** telling what would happen if this argument were used as a command name. It reports on keywords, aliases, built-ins, and functions. If the command is none of the these, it follows the path search rules and prints the full path-name, if any; otherwise, it prints an error message.

Additional Test Operators

The binary operators **–ot** and **–nt** can be used to compare the modification times of two files to see which file is *older than* or *newer than* the other. The binary operator **–ef** is used to see if two files have the same device and i-node number, i.e., a link to the same file.

The unary operator **–L** returns true for a symbolic link.

Added Trap

All traps can be given by name in **ksh**. The names of traps corresponding to signals are the same as the signal name with the **SIG** prefix removed. The trap **0** is named **EXIT** and there is also a trap named **ERR**. This trap is invoked whenever the shell would exit if the **–e** flag were set. This trap is used by Fourth Generation Make that runs **ksh** as a co-process.

Shell Accounting

There is a compile time option to the shell to generate an accounting message for each shell script.

Coded in Standard C

ksh is coded in standard C. It tries to adapt itself to the environment when it is compiled taking advantages of the features of the host environment when possible. There are far fewer *lint* messages from **ksh** then for the SYSTEM V/88 shell. **ksh** does not catch the segmentation violation signal, SIGSEGV, so that it can run on machines that cannot recover from these traps.

Internalization

ksh treats eight bit characters transparently without stripping the leading bit. There is also a compile time switch to enable handling multi-byte and multi-width characters sets.

No special meaning for ^

The SYSTEM V/88 shell uses ^ as an archaic synonym for |. The ^ is not a special character to **ksh**.

Added Conveniences

You can refer to multi-digit positional parameters in **ksh** by putting the number in braces. Thus, **${12}** is legal in **ksh** but illegal in the SYSTEM V/88 shell.

ksh performs file name expansion of file name arguments if the expansion is unique. Thus, **cat < file*** expands the file name if the expansion is unique.

If you invoke the shell as **ksh script** then **ksh** does a path search on script.

Unbalanced quotes causes the shell to print an error message giving the type of quote and the line number on which the opening quote occurs.

Run time error messages detected by the shell prints the line number within a function or script where the error was detected.

Performance

ksh executes many scripts faster than the SYSTEM V/88 shell. One major reason is that many of the functions provided by **echo**(1) and **expr**(1) are built-in. The time to execute a built-in function is one or two orders of magnitude faster than performing a fork and execute of the shell. Command substitution of built-ins is performed without creating another process, and often without even creating a temporary file.

Another reason for improved performance is that all I/O is buffered. Output buffers are flushed only when required. Several of the internal algorithms are changed so that the number of subroutine calls is substantially reduced. **ksh** uses hash tables for variables. Scripts that rely on referencing variables execute faster. More processing is performed while reading the script so that execution time is saved while running loops.

Scripts that do little internal processing and create many processes may run a little slower on SYSTEM V/88 because the time to fork **ksh** is slightly slower than for the SYSTEM V/88 shell. On BSD UNIX, **ksh** can be compiled with a **VFORK** option that uses *vfork* whenever possible. Here, binary programs startup somewhat faster but shell script files start a little slower because a separate invocation of the **ksh** is required.

The **ENV** file can have an undesirable effect on performance. Even if this file is small, the shell must perform an open of this file. If large functions are placed in the **ENV** file, they must be read in and compiled even if they are never referenced. If you only need the startup file for interactive shells set your ENV variable to a value that evaluates to a file name for interactive shells, otherwise, to the null string. If you export the startup file name in the variable **START**, the following setting only invokes the startup file for interactive shells because the subscript evaluates to 0 only if the shell is interactive.

```
ENV='${START[(_$-==1)+(_==0)-(_$-!=_${-%%*i*})]}'
```

If you need a startup **ENV** file for all shells, use a *case* statement on the **$-** parameter to distinguish which actions only apply to interactive shells. The **ENV** file should look like the following:

```
# options aliases and functions for all shell invocations
case    $- in
*i*)
        # options aliases and functions for interactive only
        ;;
esac
```

If there are functions that are only occasionally referenced, put them into a separate file **$HOME/functions** or any name you prefer and put aliases in the **ENV** file for each function name of the form.

alias *function_name*='. **$HOME/functions**;*function_name*'

In the beginning of the **$HOME/functions** file, you must unalias each function name defined in the file. The first reference to any *function_name* in the function file causes the function file to get read in and the functions compiled.

Example

The following is an example of a **ksh** script. This program is a variant of the SYSTEM V/88 **grep**(1) program. Pattern matching for this version of **grep** means shell patterns consisting of **?** , * , and **[ ]** .

The first half examines option flags. Note that all options except **–b** have been implemented. The second half goes through each line of each file to look for a pattern match.

This program is not intended to serve as a replacement for **grep**; just as an illustration of the programming power of **ksh**. Note that no auxiliary processes are spawned by this script. It was written and debugged in under two hours. While performance is acceptable for small programs, this program runs at only one tenth the speed of **grep** for large files.

```
#
#       SHELL VERSION OF GREP
#
vflag= xflag= cflag= lflag= nflag=
set -f
while   ((1))                          # look for grep options
do      case  "$1" in
        -v*)  vflag=1;;
        -x*)  xflag=1;;
        -c*)  cflag=1;;
        -l*)  lflag=1;;
        -n*)  nflag=1;;
        -b*)  print 'b option not supported';;
        -e*)  shift;expr="$1";;
        -f*)  shift;expr=$( < $1 );;
        -*)   print $0: 'unknown flag';exit 2;;
        *)    if        test "$expr" = ''
              then      expr="$1";shift
```

```
            fi
            test "$xflag" || expr="*${expr}*"
            break;;
      esac
      shift                          # next argument
done
noprint=$vflag$cflag$lflag           # don't print if these flags set
integer n=0 c=0 tc=0 nargs=$#        # initialize counters
for i in "$@"                        # go through the files
do    if     ((nargs<=1))
      then   fname=''
      else   fname="$i":
      fi
      test "$i"  &&  exec 0< $i      # open file if necessary
      while  read -r line            # read in a line
      do     let n=n+1
             case   "$line" in
             $expr)                  # line matches pattern
                    if      test "$noprint" = ""
                    then    print -r "$fname${nflag:+$n:}$line"
                    fi
                    let c=c+1 ;;
             *)                      # not a match
                    if      test "$vflag"
                    then    print -r "$fname${nflag:+$n:}$line"
                    fi;;
             esac
      done
      if     test "$lflag" && ((c))
      then   print - $i
      fi
      let tc=tc+c n=0 c=0
done
test "$cflag" && print $tc           #  print count if cflag is set
let tc                               #  set the exit value
```

A Summary of the File System

This appendix summarizes the description of the file system given in Chapter 1 and reviews the major system directories in the **root** directory.

File System Structure

The SYSTEM V/88 file system is organized in a hierarchy; its structure is often described as an inverted tree. At the top of this tree is the **root** directory, the source of the entire file system. It is designated by a / (slash). All other directories and files descend and branch out from **root**, as shown in Figure A-1.

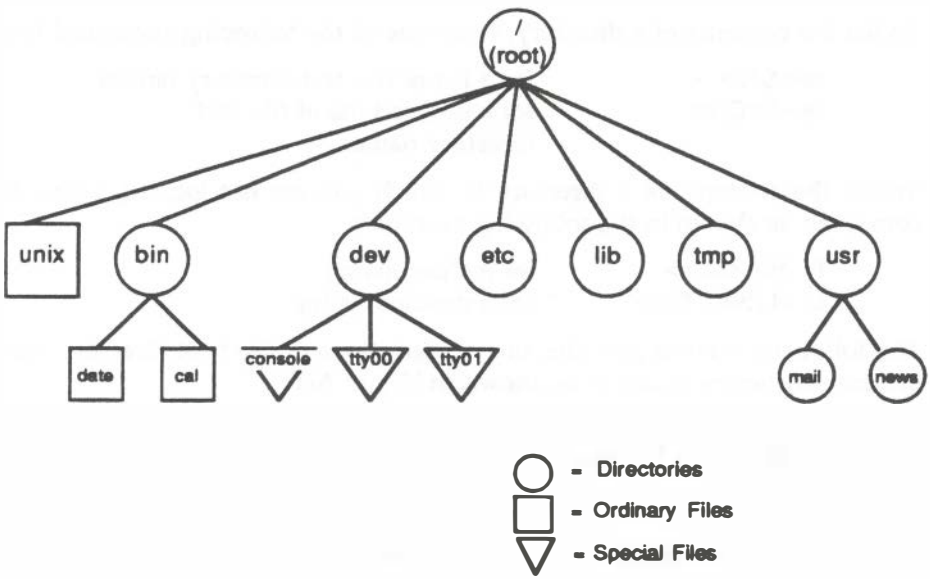


Figure A-1. Directory Tree from **root**

One path from **root** leads to your home directory. You can organize and store information in your own hierarchy of directories and files under your home directory.

Other paths lead from **root** to system directories that are available to all users. The system directories described in this book are common to all SYSTEM V/88 installations and are provided and maintained by the operating system.

In addition to this standard set of directories, your system may have other system directories. To obtain a listing of the directories and files in the **root** directory on your system, type the following command line:

ls –l /<CR>

To move around in the file structure, you can use path names. For example, you can move to the directory **/bin** (which contains system executable files) by typing the following command line:

cd /bin<CR>

To list the contents of a directory, issue one of the following command lines:

ls<CR>	for a list of file and directory names
ls –l<CR>	for a detailed list of file and directory names

To list the contents of a directory in which you are not located, issue the **ls** command as shown in the following examples:

ls /bin<CR>	for a short listing
ls –l /bin<CR>	for a detailed listing

The following section provides brief descriptions of the **root** directory and the system directories under it, as shown in Figure A-1.

SYSTEM V/88 Directories

/
The source of the file system (called **root** directory).

/bin
Contains many executable programs and utilities:

cat
date
login
grep
mkdir
who

/lib

Contains available program libraries and language libraries:

libc.a	system calls, standard I/O
libm.a	math routines and support for languages, e.g., C and FORTRAN.

/dev

Contains special files that represent peripheral devices:

console	console
lp	line printer
tty*n*	user terminal(s)
dsk/*	disks

/etc

Contains programs and data files for system administration.

/stand

Contains standalone programs, including the copy of the kernel (**sysv68**) loaded by the disk-based boot loader.

/tmp

Contains temporary files, e.g., the buffers created for editing a file.

/usr

Contains the following subdirectories which, in turn, contain the data listed:

news	important news items
mail	electronic mail
spool	files waiting to be printed on the line printer

B Summary of SYSTEM V/88 Commands

Basic SYSTEM V/88 Commands

at

Request to run a command in background mode at a time you specify on the command line. If you do not specify a time, **at**(1) displays the job numbers of all jobs you have running in **at**(1), **batch**(1), or background mode. For example:

at 8:45am Jun 09<**CR**>
command1<**CR**>
command2<**CR**>
<^**d**>

If you use the **at** command without the date, the command executes within 24 hours at the time specified.

banner

Display a message (in words up to 10 characters long) in large letters on the standard output.

batch

Submit command(s) to be processed when the system load is at an acceptable level. For example:

batch<**CR**>
command1<**CR**>
command2<**CR**>
<^**d**>

You can use a shell script for a command in **batch**(1). This may be useful and timesaving if you have a set of commands you frequently submit using this command.

cat

Display the contents of a specified file at your terminal. To halt the output on an ASCII terminal temporarily, use <^**s**>. Type <^**q**> to restart the output. To interrupt the output and return to the shell on an ASCII terminal, press the BREAK or DELETE key.

cd
Change directory from the current one to your home directory. If you include a directory name, changes from the current directory to the directory specified. By using a path name in place of the directory name, you can jump several levels with one command.

cp
Copy a specified file into a new file, leaving the original file intact.

cut
Cut out specified fields from each line of a file. This command can be used to cut columns from a table, for example.

date
Display the current date and time.

diff
Compare two files. The **diff**(1) command reports which lines are different and what changes should be made to the second file to make it the same as the first file.

echo
Display input on the standard output (the terminal), including the carriage return, and returns a prompt.

ed
Edit a specified file using the line editor. If there is no file by the name specified, the **ed**(1) command creates one. See Chapter 5 for detailed instructions on using the **ed**(1) editor.

grep
Search a specified file(s) for a specified pattern and prints those lines that contain the pattern. If you name more than one file, **grep**(1) prints the file that contains the pattern.

kill
Terminate a background process specified by its process identification number (PID). You can obtain a PID by running the **ps**(1) command.

lex
Generate programs to be used in simple lexical analysis of text, perhaps as a first step in creating a compiler. See the *User's Reference Manual* for details.

lp

Print the contents of a specified file on a line printer, giving you a paper copy of the file.

lpstat

Display the status of any requests made to the line printer. Options are available for requesting more detailed information.

ls

List the names of all files and directories except those whose names begin with a dot (**.**). Options are available for listing more detailed information about the files in the directory. See the **ls**(1) entry in the *User's Reference Manual* for details.

mail

Display any electronic mail you may have received at your terminal, one message at a time. Each message ends with **?** prompt; **mail**(1) waits for you to request an option, e.g., saving, forwarding, or deleting a message. To obtain a list of the available options, type **?**.

When followed by a login name, **mail**(1) sends a message to the owner of that name. You can type as many lines of text as you want. Then type <^**d**> to end the message and send it to the recipient. Press the **BREAK** key to interrupt the mail session.

mailx

mailx(1) is a more sophisticated, expanded version of electronic mail.

make

Maintain and support large programs or documents on the basis of smaller ones. See the **make**(1) page in the *User's Reference Manual* for details.

mkdir

Make a new directory. The new directory becomes a subdirectory of the directory in which you issue the **mkdir** command. To create subdirectories or files in the new directory, you must first move into the new directory with the **cd** command.

mv

Move a file to a new location in the file system. You can move a file to a new file name in the same directory or to a different directory. If you move a file to a different directory, you can use the same file name or choose a new one.

nohup
Place execution of a command in the background, so it continues executing after you log off of the system. Error messages are placed in a file called **nohup.out**.

pg
Display the contents of a specified file on your terminal one page at a time. After each page, the system pauses and waits for your instructions before proceeding.

pr
Display a partially formatted version of a specified file at your terminal. The **pr**(1) command shows page breaks, but does not implement any macros supplied for text formatter packages.

ps
Display the status and number of every process currently running. The **ps**(1) command does not show the status of jobs in the **at**(1) or **batch**(1) queues, but it includes these jobs when they are executing.

pwd
Display the full path name of the current working directory.

rm
Remove a file from the file system. You can use metacharacters with the **rm**(1) command but should use them with caution; a removed file cannot be recovered easily.

rmdir
Remove a directory. You cannot be in the directory you want to delete. Also, the command does not delete a directory unless it is empty. Therefore, you must remove any subdirectories and files that remain in a directory before running this command on it. See **rm -r** in the *User's Reference Manual* for the ability to remove directories that are not empty.

sort
Sort a file in ASCII order and displays the results on your terminal. ASCII order is:

1. numbers before letters
2. uppercase before lowercase
3. alphabetical order

There are other options for sorting a file. For a complete list of **sort**(1) options, see the **sort**(1) page in the *User's Reference Manual*.

spell
Collect words from a specified file and check them against a spelling list. Words not on the list or not related to words on the list (e.g., with suffixes, prefixes) display.

stty
Report the settings of certain input/output options for your terminal. When issued with the appropriate options and arguments, **stty**(1) also sets these input/output option. See the **stty**(1) entry in the *User's Reference Manual*.

uname
Display the name of the operating system on which you are currently working.

uucp
Send a specified file to another system. See the **uucp**(1) page in the *User's Reference Manual* for details.

uuname
List the names of remote operating system that can communicate with your system.

uupick
Search the public directory for files sent to you by the **uuto**(1) command. If a file is found, **uupick**(1) displays its name and the system it came from, and prompts you (with a **?**) to take action.

uustat
Report the status of the **uuto**(1) command you issued to send files to another user.

uuto
Send a specified file to another user. Specify the destination in the format *system*!*login*. The *system* must be on the list of systems generated by the **uuname**(1) command.

vi
Edit a specified file using the **vi**(1) screen editor. If there is no file by the name you specify, **vi**(1) creates one. See Chapter 6 for detailed information on using the **vi**(1) editor.

wc
Count the number of lines, words, and characters in a specified file and display the results on your terminal.

who

Display the login names of the users currently logged in on your system. List the terminal address for each login and the time each user logged in.

yacc

Impose a structure on the input of a program. See the *User's Reference Manual* for details.

C Quick Reference to ed Commands

The ed Commands

The general format for **ed** commands is:

[address1,address2]command[parameter]...<**CR**>

where *address1* and *address2* denote line addresses; the *parameters* show the data on which the command operates. The commands appear on your terminal as you type them. You can find complete information on using **ed** commands in Chapter 5, *Line Editor Tutorial.*

The following is a summary of **ed** commands. They are grouped according to function.

Commands for Getting Started

ed *filename*
Accesses the **ed** line editor to edit a specified file.

a
Appends text after the current line.

.
Ends the text input mode and returns to the command mode.

p
Displays the current line.

d
Deletes the current line.

<**CR**>
Moves down one line in the buffer.

–
Moves up one line in the buffer.

w
Writes the buffer contents to the file currently associated with the buffer.

q
Ends an editing session. If changes to the buffer were not written to a file, a warning (**?**) is issued. Typing **q** a second time ends the session without writing to a file.

Line Addressing Commands

1, 2, 3...
Denotes line addresses in the buffer.

.
Address of the current line in the buffer.

.=
Displays the current line address.

$
Denotes the last line in the buffer.

,
Addresses the first through the last line.

;
Addresses the current line through the last line.

+*x*
Relative address, determined by adding *x* to the current line number.

−*x*
Relative address, determined by subtracting *x* from the current line number.

/abc
Searches forward in the buffer and addresses the first line after the current line that contains the pattern *abc*.

?*abc*
Searches backward in the buffer and addresses the first line before the current line that contains the pattern *abc*.

g/*abc*
Addresses all lines in the buffer that contain the pattern *abc*.

v/*abc*
Addresses all lines in the buffer that do not contain the pattern *abc*.

Display Commands

p
Displays the specified lines in the buffer.

n
Displays the specified lines preceded by their line addresses and a tab space.

Text Input

a
Enters text after the specified line in the buffer.

i
Enters text before the specified line in the buffer.

c
Replaces text in the specified lines with new text.

.
When typed on a line by itself, ends the text input mode and returns to the command mode.

Deleting Text

d
Deletes one or more lines of text (command mode).

u
Undoes the last command given (command mode).

CKILL
Deletes the current line (in text input mode) or a command line (in command mode).

CERASE
Deletes the last character entered as text (in input mode).

Substituting Text

*address1,address2***s**/*old_text*/*new_text*/*command*
Substitutes *new_text* for *old_text* within the range of lines denoted by *address1,address2* (which may be numbers, symbols, or text). The *command* may be **g**, **l**, **n**, **p**, or **gp**.

Special Characters

.
Matches any single character in search or substitution patterns.

*****
Matches zero or more occurrences of the preceding character in search or substitution patterns.

[...]
Matches the first occurrence of a pattern in the brackets.

[^...]
Matches the first occurrence of a character that is not in the brackets.

.*
Matches zero or more occurrences of any characters following the period in search or substitution patterns.

^
The circumflex (^) matches the beginning of the line in search or substitution patterns.

$
Matches the end of the line in search or substitution patterns.

****
Takes away the special meaning of the special character that follows in search and substitution patterns.

&
Repeats the last pattern to be substituted.

%
Repeats the last replacement pattern.

Text Movement Commands

m
Moves the specified lines of text after a destination line; deletes the lines at the old location.

t
Copies the specified lines of text and places the copied lines after a destination line.

j
Joins the current line with the next contiguous line.

w
Copies (writes) the buffer contents into a file.

r
Reads in text from another file and appends it to the buffer.

Other Useful Commands and Information

h
Displays a short explanation for the preceding diagnostic response (**?**).

H
Turns on the help mode, which automatically displays an explanation for each diagnostic response (**?**) during the editing session.

l
Displays nonprinting characters in the text.

f
Displays the current file name.

f *newfile*
Changes the current file name associated with the buffer to *newfile*.

!*command*
Allows you to escape, temporarily, to the shell to execute a shell command.

ed.hup
If the terminal is hung up before a write command, the editing buffer is saved in the file *ed.hup*.

D Quick Reference to vi Commands

vi Quick Reference

This appendix is a glossary of commands for the screen editor **vi**. The commands are grouped according to function.

The general format of a **vi** command is:

[*x*][*command*]*text-object*

where *x* denotes a number and *text-object* shows the portion of text on which the command operates. The commands appear on your screen as you type them. For an introduction to the use of **vi** commands, see Chapter 6, *Screen Editor Tutorial*.

Commands for Getting Started

Shell Commands

TERM=*code*
Puts a code name for your terminal into the variable **TERM**.

export TERM
Conveys the value of **TERM** (the terminal code) to any SYSTEM V/88 system program that is terminal dependent.

tput init
Initializes the terminal so that it will function properly with various SYSTEM V/88 system programs.

NOTE

Before you can use **vi**, you must complete the first three steps represented by the above three lines: setting the **TERM** variable, exporting the value of **TERM**, and running the **TermSetup** command. These steps are now normally done for you by the default **.profile** provided by your administrator.

vi *filename*
Accesses the **vi** screen editor so that you can edit a specified file.

Basic vi Commands

<a>
Enters text input mode and appends text after the cursor.

<ESC>
Escape; leaves text input mode and returns to command mode.

<h>
Moves the cursor to the left one character.

<j>
Moves the cursor down one line in the same column.

<k>
Moves the cursor up one line in the same column.

<l>
Moves the cursor to the right one character.

<x>
Deletes the current character.

<CR>
Carriage return; moves the cursor down to the beginning of the next line.

<ZZ>
Writes changes made to the buffer to the file and quits **vi**.

:w
Writes changes made to the buffer to the file.

:q
Quits **vi** if changes made to the buffer have been written to a file.

Commands for Positioning in the Window

Positioning by Character

<h>
Moves the cursor one character to the left.

<BACKSPACE>
Backspace; moves the cursor one character to the left.

<l>
Moves the cursor one character to the right.

<space bar>
Moves the cursor one character to the right.

<f*x*>
Moves the cursor right to the specified character *x*.

<F*x*>
Moves the cursor left to the specified character *x*.

<t*x*>
Moves the cursor right to the character just before the specified character *x*.

<T*x*>
Moves the cursor left to the character just after the specified character *x*.

<;>
Continues the search for the character specified by the **<f>**, **<F>**, **<t>**, or **<T>** commands. The **;** remembers the character specified and searches for the next occurrence of it on the current line.

<,>
Continues the search for the character specified by the **<f>**, **<F>**, **<t>**, or **<T>** commands. The **,** remembers the character specified and searches for the previous occurrence of it on the current line.

Positioning by Line

<j>
Moves the cursor down one line from its present position, in the same column.

<k>
Moves the cursor up one line from its present position, in the same column.

<+>
Moves the cursor down to the beginning of the next line.

<CR>
Carriage return; moves the cursor down to the beginning of the next line.

<–>
Moves the cursor up to the beginning of the next line.

Positioning by Word

<w>
Moves the cursor to the right, to the first character in the next word.

<b>
Moves the cursor back to the first character of the previous word.

<e>
Moves the cursor to the end of the current word.

Positioning by Sentence

<(>
Moves the cursor to the beginning of the sentence.

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

Positioning by Paragraph

<{>
Moves the cursor to the beginning of the paragraph.

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

Positioning in the Window

<H>
Moves the cursor to the first line on the screen, or "home."

<M>
Moves the cursor to the middle line on the screen.

<L>
Moves the cursor to the last line on the screen.

Commands for Positioning in the File

Scrolling

<^f>
Scrolls the screen forward a full window, revealing the window of text below the current window.

<^d>
Scrolls the screen down a half window, revealing lines of text below the current window.

<^b>
Scrolls the screen back a full window, revealing the window of text above the current window.

<^u>
Scrolls the screen up a half window, revealing the lines of text above the current window.

Positioning on a Numbered Line

<G>
Moves the cursor to the beginning of the last line in the buffer.

<*n*G>
Moves the cursor to the beginning of the *n*th line of the file (*n* = line number).

Searching for a Pattern

/pattern
Searches forward in the buffer for the next occurrence of the *pattern* of text. Positions the cursor under the first character of the pattern.

?pattern
Searches backward in the buffer for the first occurrence of *pattern* of text. Positions the cursor under the first character of the pattern.

<n>
Repeats the last search command.

<N>
Repeats the search command in the opposite direction.

Commands for Inserting Text

<a>
Enters text input mode and appends text after the cursor.

<i>
Enters text input mode and inserts text before the cursor.

<o>
Enters text input mode by opening a new line immediately below the current line.

<O>
Enters text input mode by opening a new line immediately above the current line.

<ESC>
Escape; returns to command mode from text input mode (entered with any of the above commands).

Commands for Deleting Text

In Text Input Mode

<BACKSPACE>
Backspace; deletes the current character.

<^w>
Deletes the current word delimited by blanks.

CKILL
Erases the current line of text.

In Command Mode

<x>
Deletes the current character.

<dw>
Deletes a word (or part of a word) from the cursor through the next space or to the next punctuation.

<dd>
Deletes the current line.

<*n*d*x*>
Deletes *n* number of text objects of type *x*, where *x* may be as a word, line, sentence, or paragraph.

<D>
Deletes the current line from the cursor to the end of the line.

Commands for Modifying Text

Characters, Words, Text Objects

<r>
Replaces the current character.

<s>
Deletes the current character and appends text until the **<ESC>** command is typed.

<S>
Replaces all the characters in the current line.

<~>
Changes uppercase to lowercase or lowercase to uppercase.

<cw>
Replaces the current word or the remaining characters in the current word with new text, from the cursor to the next space or punctuation.

<cc>
Replaces all the characters in the current line.

<*n*c*x*>
Replaces *n* number of text objects of type *x*, where *x* may be a word, line, sentence, or paragraph.

<C>
Replaces the remaining characters in the current line, from the cursor to the end of the line.

Cutting and Pasting Text

<p>
Places the contents of the temporary buffer (containing the output of the last delete or yank command) into the text after the cursor or below the current line.

<yy>
ks (extracts) a specified line of text and puts it into a temporary buffer.

<*n*y*x*>
Extracts a copy of *n* number of text objects of type *x* and puts it into a temporary buffer.

<"*l*y*x*>
Places a copy of text object *x* into a register named by a letter *l*. *x* may be a word, line, sentence, or paragraph.

<"*x*p>
Places the contents of register *x* after the cursor or below the current line.

Other Commands

Special Commands

<^g>
Gives the line number of current cursor position in the buffer and modification status of the file.

<.>
eats the action performed by the last command.

<u>
Undoes the effects of the last command.

<U>
Restores the current line to its state prior to present changes.

<J>
Joins the line immediately below the current line with the current line.

<^l>
Clears and redraws the current window.

Line Editor Commands

:
Tells **vi** that the next commands you issue will be line editor commands.

:sh
Temporarily returns to the shell to perform some shell commands without leaving **vi**.

<^d>
Escapes the temporary return to the shell and returns to **vi** so you can edit the current window.

:*n*
Goes to the *n*th line of the buffer.

:*x*,*z***w** *filename*
Writes lines from the number *x* through the number *z* into a new file called *filename.*

:$
Moves the cursor to the beginning of the last line in the buffer.

:.,$d
Deletes all lines from the current line to the last line.

:r *filename*
Inserts the contents of the file *filename* under the current line of the buffer.

:s/*text***/***new_text***/**
Replaces the first instance of *text* on the current line with *new_text*.

:s/*text***/***new_text***/g**
Replace every occurrence of *text* on the current line with *new_text*.

:g/*text***/s//***new_text***/g**
Changes every occurrence of *text* in the buffer to *new_text*.

Commands for Quitting vi

<ZZ>
Writes the buffer to the file and quits **vi**.

:wq
Writes the buffer to the file and quits **vi**.

:w *filename*
:q
Writes the buffer to the new file *filename* and quits **vi**.

:w! *filename*
:q
Overwrites the existing file *filename* with the contents of the buffer and quits **vi**.

:q!
Quits **vi** whether or not changes made to the buffer were written to a file. Does not incorporate changes made to the buffer since the last write (**:w**) command.

:q
Quits **vi** if changes made to the buffer were written to a file.

Special Options for vi

vi *file1 file2 file3*
Enters three files into the **vi** buffer to be edited. Those files are *file1*, *file2*, and *file3*.

:w
:n
When more than one file has been called on a single **vi** command line, writes the buffer to the file you are editing and then calls the next file in the buffer (use **:n** only after **:w**).

vi −r *file1*
Restores the changes made to *file1* that were lost because of an interrupt in the system.

view *file1*

Displays *file1* in the read-only mode of **vi**. Any changes made to the buffer cannot be written to the file.

E Summary of Shell Command Language

Summary of Shell Command Language

This appendix is a summary of the shell command language and programming constructs discussed in Chapter 7, *Shell Tutorial*. The first section reviews metacharacters, special characters, input and output redirection, variables and processes. These are arranged by topic in the order that they were discussed in the chapter. The second section contains models of the shell programming constructs.

The Vocabulary of Shell Command Language

Special Characters in the Shell

*** ? [ ] .**
Metacharacters; used to provide a shortcut to referencing file names, through pattern matching.

&
Executes commands in the background mode.

;
Sequentially executes several commands typed on one line, each pair separated by ;.

****
Turns off the meaning of the immediately following special character.

'...'
Enclosing single quotes turn off the special meaning of all characters.

"..."
Enclosing double quotes turn off the special meaning of all characters except **$** and **`** .

Redirecting Input and Output

<
Redirects the contents of a file into a command.

>
Redirects the output of a command into a new file, or replaces the contents of an existing file with the output.

>>
Redirects the output of a command so that it is appended to the end of a file.

|
Directs the output of one command so that it becomes the input of the next command.

`*command*`
Substitutes the output of the enclosed command in place of `*command*`.

Executing and Terminating Processes

batch
Submits the following commands to be processed at a time when the system load is at an acceptable level. <^**d**> ends the **batch** command.

at
Submits the following commands to be executed at a specified time. <^**d**> ends the **at** command.

at –l
Reports which jobs are currently in the **at** or **batch** queue.

at –r
Removes the **at** or **batch** job from the queue.

ps
Reports the status of the shell processes.

kill *PID*
Terminates the shell process with the specified process ID (PID).

nohup *command list* **&**
Continues background processes after logging off.

CINTR
This key (default DELETE) terminates most interactive commands.

Making a File Accessible to the Shell

chmod u+x *filename*
Gives the user permission to execute the file (useful for shell program files).

mv *filename* **$HOME/bin/***filename*
Moves your file to the **bin** directory in your home directory. This **bin** holds executable shell programs that you want to be accessible. Make sure the **PATH** variable in your **.profile** file specifies this **bin**. If it does, the shell searches in **$HOME/bin** for your file when you try to execute it. If your **PATH** variable does not include your **bin**, the shell does not know where to find your file and your attempt to execute it fails.

filename
The name of a file that contains a shell program becomes the command that you type to run that shell program.

Variables

positional parameter
A numbered variable used within a shell program to reference values automatically assigned by the shell from the arguments of the command line invoking the shell pro

echo
A command used to print the value of a variable on your terminal.

$#
A special parameter that contains the number of arguments with which the shell program has been executed.

$*
A special parameter that contains the values of all arguments with which the shell program has been executed.

named variable
A variable to which the user can give a name and assign values.

Variables Used in the System

HOME
Denotes your home directory; the default variable for the **cd** command.

PATH
Defines the path your login shell follows to find commands.

CDPATH
Defines the search path for the **cd** command.

MAIL
Gives the name of the file containing your electronic mail.

PS1 PS2
Define the primary and secondary prompt strings.

TERM
Defines the type of terminal.

LOGNAME
Login name of the user.

IFS
Defines the internal field separators (normally the space, the tab, and the carriage return).

TERMINFO
Allows you to request that the **curses** and **terminfo** subroutines search a specified directory tree before searching the default directory for your terminal type.

TZ
Sets and maintains the local time zone.

Shell Programming Constructs

here Document

```
command <<!
input lines
!
```

for loop

```
for variable<CR>
        in this list of values<CR>
do the following commands<CR>
        command 1<CR>
        command 2<CR>
        last command<CR>
done<CR>
```

while loop

```
while command list<CR>
do<CR>
    command1<CR>
    command2<CR>
    last command<CR>
done<CR>
```

if...then

```
if this command is successful<CR>
then command1<CR>
    command2<CR>
    last command<CR>
fi<CR>
```

if...then...else

```
if command list<CR>
    then command list<CR>
    else command list<CR>
fi<CR>
```

The test Command for Loops

The **test** command, which checks to see if certain conditions are true, is a useful command for conditional constructs. If the condition is true, the loop continues. If the condition is false, the loop ends and the next command is executed. Some of the useful options for the **test** command are:

test **−r** *file*<**CR**>
true if the file exists and is readable

test **−w** *file*<**CR**>
true if the file exists and has write permission

test **−x** *file*<**CR**>
true if the file exists and is executable

test **−s** *file*<**CR**>
true if the file exists and has at least one character

test *var1* **−eq** *var2*<**CR**>
true if *var1* equals *var2*

test *var1* **−ne** *var2*<**CR**>
true if *var1* does not equal *var2*

case Construction

```
case word<CR>
in<CR>
    pattern1)<CR>
        command line 1<CR>
        last command line<CR>
    ;;<CR>
    pattern2)<CR>
        command line 1<CR>
        last command line<CR>
    ;;<CR>
    pattern3)<CR>
        command line 1<CR>
        last command line<CR>
    ;;<CR>
esac<CR>
```

break and continue Statements

A break or continue statement forces the program to leave any loop and execute the command following the end of the loop.

F Setting Terminal Type

Setting the TERM Variable

Several types of terminals are supported by SYSTEM V/88 systems. Because some commands are terminal dependent, the system must know what type of terminal you are using whenever you log in. The system determines the characteristics of your terminal by checking the value of a variable called **TERM** which holds the name of a terminal. If you placed the name of your terminal into this variable, the system can execute all programs in a way that is suitable for your terminal.

This method of telling the system what type of terminal you are using is called setting the terminal configuration. To set your terminal configuration, type the command lines shown on the following screen, substituting the name of your terminal for *terminal_name*:

```
$ TERM=terminal_name<CR>
$ export TERM<CR>
$ tput init<CR>
```

These lines must be executed in the order shown; otherwise, they will not work. Also, this procedure must be repeated every time you log in. Therefore, most users put these lines into the **.profile** file that is automatically executed every time they log in. For details about the **.profile** file, see Chapter 7.

The first two lines in the screen tell the system shell what type of terminal you are using. The **tput init** command line instructs your terminal to behave in ways that the system expects a terminal of that type to behave. For example, it sets the terminals left margin and tabs, if those capabilities exist for the terminal.

The **tput** command uses the entry in this database for your terminal to make terminal dependent capabilities and information available to the shell. Because the values of these capabilities differ for each type of terminal, you must execute the **tput init** command line every time you change the **TERM** variable.

For each terminal type, a set of capabilities is defined in a database. This database is usually found in either the **/usr/lib/terminfo** or **/usr/lib/.COREterm** directory, depending on the system.

NOTE

Every system has at least one of these directories; some may have both. Your System Administrator can tell you whether your system has the **terminfo** and/or the **.COREterm** directory.

The following sections describe how you can determine what *terminal_names* are acceptable. Further information about the capabilities in the **terminfo** database can be found on the **terminfo(4)** manual page in the *Programmer's Reference Manual*.

Acceptable Terminal Names

The operating system recognizes a wide range of terminal types. Before you put a terminal name into the **TERM** variable, you must make sure that your terminal is one of them.

You must also verify that the name you put into the **TERM** variable is a recognized terminal name. There are usually at least two recognized names: name of the manufacturer and the model number. However, there are several ways to represent these names; e.g., by varying the use of uppercase and lowercase, using abbreviations. Do not put a terminal name in the **TERM** variable until you have verified that the system recognizes it.

The **tput** command provides a quick way to make sure your terminal is supported by your system, type:

tput –T*terminal_name* **longname**<CR>

If your system supports your terminal, it responds with the complete name of your terminal. Otherwise, you get an error message.

To find an acceptable name that you can put in the **TERM** variable, find a listing for your terminal in the directory **/usr/lib/terminfo** or **/usr/lib/.COREterm**. Each of these directories is a collection of directories with single-character names. Each directory holds a list of terminal names that all begin with the name of the directory. This name can be either a letter, e.g., the initial U in UniSoft, or a number, e.g., the initial 5 in 5425. Find the directory whose name matches the first character of your terminal's name. Then list the directories contents and look for your terminal, e.g., **/usr/lib/terminfo/v/vt100**.

You can also check with your System Administrator for a list of terminals supported by your system and the acceptable names you can put in the **TERM** variable.

Example

The terminal you want to set up is an AT&T Teletype Model 5425. Your login is **jim** and you are currently in your home directory. First, you verify that your system supports your terminal by running the **tput** command. Next, you find an acceptable name for it in the **/usr/lib/terminfo/A** directory. The following screen shows which commands you need to do this:

```
$ tput –T5425 longname<CR>
AT&T 4425/5425
$ cd /usr/lib/.COREterm/A<CR>
$ ls
ATT4410
ATT4415
ATT4418
ATT4424
ATT4424-2
ATT4425
ATT4426
ATT513
ATT5410
ATT5418
ATT5420
ATT5420-2
ATT5425
ATT5620
ATT610BCT
ATTPT505
$
```

Now you are ready to put the name you found, **ATT5425**, in the **TERM** variable. Whenever you do this, you must also export **TERM** and execute **tput init**.

```
$ TERM=ATT5425<CR>
$ export TERM<CR>
$ tput init<CR>
$
```

The system now knows what type of terminal you are using and executes commands appropriately.

The Terminal Support System

The Terminal Support System is a collection of shell files. These shell files include files meant to be executed at login time; files that set the **TERM** variable; files that initialize the terminal; and files that reassign the default erase, kill interrupt, and abort characters. The Terminal Support System may be configured to local preferences and is flexible enough to support most any terminal that can be attached to the system.

This systems consists of the following shell files that are found in **/local/bin**.

TermIs
TermSetup
TermAssume
TermFuncs

The proper use of these files is found in the default profile **/etc/stdprofile**. The terminal setup consists of several lines near the start of this file. This prototype profile is installed automatically as **.profile** when a user login is created via **sysadm adduser**. Existing users should examine **/etc/stdprofile** and extract the terminal setup section into their own **.profile**. This sequence and invocation will remain the same in future releases, so you will only have to do this once.

After the modification of the necessary **.profile** files, the terminal support system can be used. The features include:

- default terminal identification through **TermAssume**
- correct initialization of any described terminal
- login-time specification of a terminal name
- the functions **normal** to set/get line-editing values

You invoke **TermIs** to select the **TERM** name for the session. This may use an assumed terminal name, which may depend on your login name and tty port (see the discussion of **TermAssume** below). If invoked with the **ask** option, **TermIs** will prompt you to select a terminal name. The actual menu of selected terminals may be different in later releases, and may be tailored to your individual site (by editing **TermIs**). In response to the prompt, you can always enter a name exactly as you want it to be assigned to the **TERM** variable.

TermIs verifies that a valid **terminfo** entry exists for any chosen or assumed terminal type, unless the system is in Single-User mode (otherwise, **/usr** must be mounted). If no entry exists, it prompts for another name. **TermIs** returns the terminal name for assignment to **TERM**. Even if you preassign the value of **TERM** in the **.profile**, **TermIs** will verify that it is a valid terminal (and prompt if not or if given the **ask** argument).

Once a valid terminal name is selected, you should always invoke **TermSetup** to be sure the terminal is properly initialized. The correct invocation is shown in **stdprofile**. Upon return, the proper initialization strings will have been sent, any download programs run, etc., as specified in the terminal's description. This procedure requires an accurate terminfo description, therefore, the **usr** filesystem must be mounted.

The actions of **TermSetup** are almost identical to those of **tput init** except that **TermSetup** does an intermediate check of the return value from the terminal initialization program (specified by the *iprog* value from **terminfo**), before sending the **is1** string.

TermAssume is used (by **TermIs**) to set an assumed (default) terminal name. If no other terminal name is specified, the assumed value will be used (i.e., the value of the **ASSUME** shell variable). Assumptions can be made for a specific port or range of ports (direct-connect terminals), a given user on a port (dial-in or switched ports), or any other situation that may occur. **TermAssume** is edited to describe your system. It contains internal

comments that describe how to do this. The supplied description is only an example, and should be edited to suit. If you have no assumed terminals, **ASSUME** will be set, by default, to **vt100**.

When any user logs in, a terminal name can be given at that time that will then be the **TERM** value for the session, overriding any assumed value. To use this feature, enter the terminal name at the **login:** prompt:

```
login: wombat term_name
Password:
```

term_name will be verified, then the terminal will be properly initialized during the login process, but only if the terminal support system has been properly invoked as shown in **/local/stdprofile**.

The shell-functions **normal** and **gtty** are defined in the terminal setup process by **TermFuncs**. Shell-functions behave just like ordinary commands, except they are executed in the current shell.

Invoking **normal** sets the erase, kill, intr, and quit keys to appropriate values for the terminal (or the defaults) and perform any other operations necessary to set "normal" modes for the terminal/user. The values of the following shell variables override the corresponding default settings of **normal**, thus preserving consistent line editing characters regardless of terminal type:

CERASE
CKILL
CINTR
CQUIT

For example, **CERASE="^H"** would set <^h> to be the erase character whenever **normal** was invoked (such as within **.profile**).

Invoking **gtty** reports the current values of erase, ill, intr, and quit. It is invoked by **normal**, to appraise you of the current environment.

These shell functions only exist in your current (login) shell. They may be generated in sub-shells (**CERASE-CQUIT** must be exported) by the following:

. TermFuncs

/local/bin/TermFuncs should be modified for local preferences and terminal types.

NOTE

The default **.profile** for the **root** login will not invoke **TermSetup** if the system is not in Multi-User mode. Likewise, this **.profile**, by default, set the values of **CERASE**, **CKILL**, and **CINTR** to the defaults before invoking **normal**, so **root** always uses # as the erase character, @ as the kill character and **DEL** key as the intr key. This is done to maintain consistency with the erase/kill/intr parameters in effect during system bring-up and shutdown (the local administrator may, of course, override this). If the individual **rc** files are modified correctly, the erase and kill characters in effect for bring-up and shutdown can be set to local preferences. There is no automatic method, at present, to effect these changes.

In future releases, **Termls** may support an "auto-identify" feature whereby terminals identify themselves. This would use the answerback message feature available in many terminals, so you could not use this feature for anything else.

G Glossary

acoustic coupler
A device that permits transmission of data over an ordinary telephone line. When you place a telephone handset in the coupler, you link a computer at one end of the phone line to a peripheral device, such as a user terminal, at the other.

address
Generally, a number that indicates the location of information in the computer's memory. In the SYSTEM V/88 operating system, the address is part of an editor command that specifies a line number or range.

append mode
A text editing mode in which the characters you type are entered as text into the text editor's buffer. In this mode you enter (append) text after the current position in the buffer. See **text input mode**; compare with **command mode** and **insert mode**.

argument
The element of a command line that specifies data on which a command is to operate. Arguments follow the command name and can include numbers, letters, or text strings. For instance, in the command **lp −m myfile**, **lp** is the command and **myfile** is the argument. See **option**.

ASCII
(pronounced **as'-kee**) American Standard Code for Information Interchange, a standard for data transmission that is used in the UNIX system. ASCII assigns sets of 0s and 1s to represent 128 characters, including alphabetical characters, numerals, and standard special characters, such as #, $, %, and &.

background
A type of program execution where you request the shell to run a command away from the interaction between you and the computer ("in the background"). While this command runs, the shell prompts you to enter other commands through the terminal.

baud rate
A measure of the speed of data transfer from a computer to a peripheral device (e.g., as a terminal) or from one device to another. Common baud rates are 300, 1200, 4800, and 9600. As a general guide, divide a baud rate by 10 to get the approximate number of English characters transmitted each second.

buffer
A temporary storage area of the computer used by text editors to make changes to a copy of an existing file. When you edit a file, its contents are read into a buffer, where you make changes to the text. For the changes to become a part of the permanent file, you must write the buffer contents back into the file. See **permanent file.**

child directory
See **subdirectory.**

command
The name of a file that contains a program that can be executed by the computer on request. Compiled programs and shell programs are forms of commands.

command file
See **executable file.**

command language interpreter
A program that acts as a direct interface between you and the computer. In the SYSTEM V/88 operating system, a program called the **shell** takes commands and translates them into a language understood by the computer.

command line
A line containing one or more commands, ended by typing a carriage return (<CR>). The line may also contain options and arguments for the commands. You type a command line to the shell to instruct the computer to perform one or more tasks.

command mode
A text editing mode in which the characters you type are interpreted as editing commands. This mode permits actions such as moving around in the buffer, deleting text, or moving lines of text. See **text input mode**; compare with **append mode** and **insert mode.**

context search
A technique for locating a specified pattern of characters (called a string) when in a text editor. Editing commands that cause a context search scan the buffer, looking for a match with the string specified in the command. See **string**.

control character
A nonprinting character that is entered by holding down the CTRL key and typing a character. Control characters are often used for special purposes, e.g., when viewing a long file on your screen with the **cat** command, typing CTRL-s (^s) stops the display so you can read it; typing CTRL-q (^q) continues the display.

current directory
The directory in which you are presently working. You have direct access to all files and subdirectories contained in your current directory. The shorthand notation for the current directory is a dot (.).

cursor
A cue printed on the terminal screen that indicates the position at which you enter or delete a character. It is usually a rectangle or a blinking underscore character.

default
An automatically assigned value or condition that exists unless you explicitly change it. For example, the shell prompt string has a default value of $ unless you change it.

delimiter
A character that logically separates words or arguments on a command line. Two frequently used delimiters in the SYSTEM V/88 operating system are the space and the tab.

diagnostic
A message printed at your terminal to indicate an error encountered while trying to execute some command or program. Generally, you need not respond directly to a diagnostic message.

directory
A type of file used to group and organize other files or directories. You cannot directly enter text or other data into a directory. (For more detail, see Appendix A, *Summary of the File System.*)

disk
A magnetic data storage device consisting of several round plates similar to phonograph records. Disks store large amounts of data and allow quick access to any piece of data.

electronic mail
The feature of an operating system that allows computer users to exchange written messages via the computer. The **mail** command provides electronic mail in which the addresses are the login names of users.

environment
The conditions under which you work while using the SYSTEM V/88 operating system. Your environment includes those things that personalize your login and allow you to interact in specific ways with the operating system and the computer. For example, your shell environment includes such things as your shell prompt string, specifics for backspace and erase characters, and commands for sending output from your terminal to the computer.

erase character
The character you type to delete the previous character you typed. The default erase character is #; some users set the erase character to the BACKSPACE key.

escape
A means of getting into the shell from within a text editor or other program.

execute
The computer's action of running a program or command and performing the indicated operations.

executable file
A file that can be processed or executed by the computer without any further translation. When you type in the file name, the commands in the file are executed. See **shell procedure**.

file
A collection of information in the form of a stream of characters. Files may contain data, programs, or other text. You access operating system files by name. See **ordinary file, permanent file,** and **executable file**.

file name
A sequence of characters that denotes a file. (In the SYSTEM V/88 operating system, a slash character (/) cannot be used as part of a file name.)

file system
A collection of files and the structure that links them together. The SYSTEM V/88 file system is a hierarchical structure. (For more detail, see Appendix A, *Summary of the File System.*)

filter
A command that reads the standard input, acts on it in some way, and then prints the result as standard output.

final copy
The completed, printed version of a file of text.

foreground
The normal type of command execution. When executing a command in foreground, the shell waits for one command to end before prompting you for another command. In other words, you enter something into the computer and the computer replies before you enter something else.

full-duplex
A type of data communication in which a computer system can transmit and receive data simultaneously. Terminals and modems usually have settings for half-duplex (one-way) and full-duplex communication; the SYSTEM V/88 operating system uses the full-duplex setting.

full path name
A path name that originates at the root directory of the SYSTEM V/88 operating system and leads to a specific file or directory. Each file and directory in the operating system has a unique full path name, sometimes called an absolute path name. See **path name**.

global
A term that indicates the complete or entire file. While normal editor commands commonly act on only the first instance of a pattern in the file, global commands can perform the action on all instances in the file.

hardware
The physical machinery of a computer and any associated devices.

hidden character
One of a group of characters within the standard ASCII character set that are not printable. Characters such as **BACKSPACE**, **ESCAPE**, and <^d> are examples.

home directory
The directory in which you are located when you log in to the SYSTEM V/88 operating system; also known as your login directory.

input/output
The path by which information enters a computer system (input) and leaves the system (output). An input device that you use is the terminal keyboard and an output device is the terminal display.

insert mode
A text editing mode in which the characters you type are entered as text into the text editor's buffer. In this mode, you enter (insert) text before the current position in the buffer. See **text input mode**; compare with **append mode** and **command mode**.

interactive
Describes an operating system (such as the SYSTEM V/88 operating system) that can handle immediate-response communication between you and the computer. In other words, you interact with the computer from moment to moment.

kill character
The character you type to delete the current (as yet unentered) input line. The default kill character is @. Many users set it to <^u>.

line editor
An editing program in which text is operated upon on a line-by-line basis within a file. Commands for creating, changing, and removing text use line addresses to determine where in the file the changes are made. Changes can be viewed after they are made by displaying the lines changed. See **text editor**; compare with **screen editor**.

login
The procedure used to gain access to the SYSTEM V/88 operating system.

login directory
See **home directory**.

login name
A string of characters used to identify a user. Your login name is different from other login names.

log off
The procedure used to exit from the operating system.

metacharacter
A subset of the set of special characters that have special meaning to the shell. The metacharacters are *, ?, and the pair [ ]. Metacharacters are used in patterns to match file names.

mode
In general, a particular type of operation (e.g., an editor's append mode). In relation to the file system, a mode is an octal number used to determine who can have access to your files and what kind of access they can have. See **permissions**.

modem
A device that connects a terminal and a computer by way of a telephone line. A modem converts digital signals to tones and converts tones back to digital signals, allowing a terminal and a computer to exchange data over standard telephone lines.

multitasking
The ability of an operating system to execute more than one program at a time.

multi-user
The ability of an operating system to support several users on the system at the same time.

nroff
A text formatter. You can use the **nroff** program to produce a formatted online copy or a printed copy of a file. See **text formatter**. (This feature is not available with the current release.)

operating system
The software system on a computer under which all other software runs. The SYSTEM V/88 system is an operating system.

option
Special instructions that modify how a command runs. Options are a type of argument that follow a command and usually precede other arguments on the command line. By convention, an option is preceded by a minus sign (−); this distinguishes it from other arguments. You can specify more than one option for some commands given in the SYSTEM V/88 operating system. For example, in the command **ls −l −a directory**, **−l** and **−a** are options that modify the **ls** command. See **argument**.

ordinary file
A file, containing text or data, that is not executable. See **executable file**.

output
Information processed in some fashion by a computer and delivered to you by way of a printer, a terminal, or a similar device.

parameter
A special type of variable used within shell programs to access values related to the arguments on the command line or the environment in which the program is execut See **positional parameter**.

parent directory
The directory immediately above a subdirectory or file in the file system organization. The shorthand notation for the parent directory is two dots (..).

parity
A method used by a computer for checking that the data received matches the data sent.

password
A code word known only to you that is called for in the login process. The computer uses the password to verify that you may indeed use the system.

path name
A sequence of directory names separated by the slash character (/) and ending with the name of a file or directory. The path name defines the connection path between some directory and the named file.

peripheral device
Auxiliary devices under the control of the main computer, used mostly for input, output, and storage functions. Some examples include terminals, printers, and disk drives.

permanent file
The data stored permanently in the file system structure. To change a permanent file, you can use a text editor, which maintains a temporary work space, or buffer, apart from the permanent files. Once changes have been made to the buffer, they must be written to the permanent file to make the changes permanent. See **buffer**.

permissions
Access modes, associated with directories and files, that permit or deny system users the ability to read, write, and/or execute the directories and files. You determine the permissions for your directories and files by changing the mode for each one with the **chmod** command.

pipe
A method of redirecting the output of one command to be the input of another command. For example, the shell command **who | wc −l** pipes output from the **who** command to the **wc** command, telling you the total number of people logged into your operating system.

pipeline
A series of filters separated by | (the pipe character). The output of each filter becomes the input of the next filter in the line. The last filter in the pipeline writes to its standard output, or may be redirected to a file. See **filter**.

positional parameters
Numbered variables used within a shell procedure to access the strings specified as arguments on the command line invoking the shell procedure. The name of the shell procedure is positional parameter **$0**. See **variable** and **shell procedure**.

prompt
A cue displayed at your terminal by the shell, telling you that the shell is ready to accept your next request. The prompt can be a character or a series of characters. The SYSTEM V/88 operating system default prompt is the dollar sign character ($).

printer
An output device that prints the data it receives from the computer on paper.

process
Generally a program that is at some stage of execution. In the SYSTEM V/88 operating system, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current directory, status of files, information recorded at login time, and various other items.

program
The instructions given to a computer on how to do a specific task. Programs are user-executable software.

read-ahead capability
The ability of the SYSTEM V/88 operating system to read and interpret your input while sending output information to your terminal in response to previous input. The operating system separates input from output and processes each correctly.

relative path name
The path name to a file or directory which varies in relation to the directory in which you are currently working.

remote system
A system other than the one on which you are working.

root
The source directory of all files and directories in the file system; designated by the slash character (/).

screen editor
An editing program in which text is operated on relative to the position of the cursor on a visual display. Commands for entering, changing, and removing text involve moving the cursor to the area to be altered and performing the necessary operation. Changes are viewed on the terminal display as they are made. See **text editor**; compare with **line editor**.

search pattern
See **string**.

search string
See **string**.

secondary prompt
A cue displayed at your terminal by the shell to tell you that the command typed in response to the primary prompt is incomplete. The SYSTEM V/88 operating system default secondary prompt is the greater-than character (>).

shell
An operating system program that handles the communication between you and the computer. The shell is also known as a command language interpreter because it translates your commands into a language understandable by the computer. The shell accepts commands and causes the appropriate program to be executed.

shell procedure
An executable file that is not a compiled program. A shell procedure calls the shell to read and execute commands contained in a file. This lets you store a sequence of commands in a file for repeated use. It is also called a shell program or command file. See **executable file**.

silent character
See **hidden character**.

software
Instructions and programs that tell the computer what to do. Contrast with **hardware**.

source code
The uncompiled version of a program written in a language such as C or Pascal. The source code must be translated to machine language by a program known as a compiler before the computer can execute the program.

special character
A character having special meaning to the shell program and used for common shell functions, e.g., file redirection, piping, background execution, and file name expansion. The special characters include <, >, |, ;, &, *, ?, [, and].

special file
A file (called a device driver) used as an interface to an input/output device, such as a user terminal, a disk drive, or a line printer.

standard input
An open file that is normally connected directly to the keyboard. Standard input to a command normally goes from the keyboard to this file and then into the shell. You can redirect the standard input to come from another file instead of from the keyboard; use an argument in the form < **file**. Input to the command then comes from the specified file.

standard output
An open file that is normally connected directly to a primary output device, e.g., a terminal printer or screen. Standard output from the computer normally goes to this file and then to the output device. You can redirect the standard output into another file instead of to the printer or screen; use an argument in the form > **file**. Output then goes to the specified file.

string
Designation for a particular group or pattern of characters, e.g., a word or phrase, that may contain special characters. In a text editor, a context search interprets the special characters and attempts to match the specified pattern with a string in the editor buffer.

string variable
A sequence of characters that can be the value of a shell variable. See **variable**.

subdirectory
A directory pointed to by a directory one level above it in the file system organization; also called a child directory.

System Administrator
The person who monitors and controls the computer on which your operating system runs; sometimes referred to as a super-user.

SYSTEM V/88 operating system
A general-purpose, multi-user, interactive time-sharing operating system developed by Motorola, Inc. under a license from AT&T Bell Laboratories. The SYSTEM V/88 operating system allows limited computer resources to be shared by several users and efficiently organizes the user's interface to a computer system.

terminal
An input/output device connected to a computer system, usually consisting of a keyboard with a video display or a printer. A terminal allows you to give the computer instructions and to receive information in response.

text editor
Software for creating, changing, or removing text with the aid of a computer. Most text editors have two modes —an input mode for typing in text and a command mode for moving or modifying text. Two examples are the SYSTEM V/88 operating system editors **ed** and **vi**. See **line editor** and **screen editor**.

text formatter
A program that prepares a file of text for printed output. To make use of a text formatter, your file must also contain some special commands for structuring the final copy. For example, these special commands tell the formatter to justify margins, start new paragraphs, set up lists and tables, place figures. Two text formatters are **nroff** and **troff** (not available with this release).

text input mode
A text editing mode in which the characters you type are entered as text into the text editor's buffer. To execute a command, you must leave text input mode. See **command mode**; compare with **append mode** and insert mode.

timesharing
A method of operation in which several users share a common computer system seemingly simultaneously. The computer interacts with each user in sequence, but the high-speed operation makes it seem that the computer is giving each user its complete attention.

tool
A package of software programs.

troff
A text formatter. The **troff** program drives a phototypesetter to produce high-quality printed text from a file (not available with this release). See **text formatter**.

TTY
Historically, the abbreviation for a teletype terminal. Today, it is generally used to denote a user terminal.

user
Anyone who uses a computer or an operating system.

user-defined
Something determined by the user.

user-defined variable
A named variable given a value by the user. See **variable.**

utility
Software used to carry out routine functions or to assist a programmer or system user in establishing routine tasks.

variable
A symbol whose value may change. In the shell, a variable is a symbol representing some string of characters (a **string value**). Variables may be used in an interactive shell as well as within a shell procedure. Within a shell procedure, positional parameters and keyword parameters are two forms of variables.

video display terminal
A terminal that uses a television-like screen (a monitor) to display information. A video display terminal can display information much faster than printing terminals.

visual editor
See **screen editor.**

working directory
See **current directory.**

MOTOROLA INC.

Microcomputer Division
2900 South Diablo Way
Tempe, Arizona 85282
P.O Box 2953
Phoenix, Arizona 85062

11040 PRINTED IN USA (3/90) WPC 2,500