# IRIS-4D User's Guide

IRIS-4D Series

SiliconGraphics
Computer Systems

Document number: 007-0605-010

# IRIS-4D User's Guide

*Version 1.0*

Document Number 007-0605-010

**Technical Publications:**

Marcia Allen
Kathleen Chaix

Special Thanks to the Technical Marketing Group



**IRIS-4D User's Guide**
**Version 1.0**
**Document Number 007-0605-010**

**Silicon Graphics, Inc.**
**Mountain View, California**

# Contents

# Purpose

The material in this guide is organized into two major parts: an overview of the UNIX operating system and a set of tutorials on the main tools available on the UNIX system. A brief description of each part follows. The last section of this Preface, "Notation Conventions", describes the typographical notation with which all the chapters of this guide conform. You may want to refer back to this section from time to time as you read the guide.

## System Overview

This part consists of Chapters 1–3, which introduce you to the basic principles of the UNIX 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 UNIX System?", provides an overview of the operating system.
- Chapter 2, "Basics for UNIX System Users", discusses the general rules and guidelines for using the UNIX system. It covers topics related to using your terminal, obtaining a system account, and establishing contact with the UNIX system.
- Chapter 3, "Using the File System", 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.

## UNIX System Tutorials

The second part of the guide consists of tutorials on the following topics: the **vi** text editor, the **ed** text editor, the shell command language and programming language, and electronic communication tools. For a thorough understanding of the material, we recommend that you work through the examples and exercises as you read each tutorial.

- Chapter 4, "Overview of the Tutorials", introduces the four chapters of tutorials in the second half of the guide. It highlights UNIX system capabilities such as command execution, text editing, electronic communication, programming, and aids to software development.
- Chapter 5, "Screen Editor Tutorial (**vi**)", teaches you how to use the visual text editor, **vi**, to create and modify text on a video display terminal.

- Chapter 6, "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.

NOTE

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

- Chapter 7, "The Bourne Shell", teaches you to how to use the Bourne shell, both as a command interpreter and as a programming language, to create shell programs. At the end of this chapter you will find an appendix describing the most commonly used **ed** commands.
- Chapter 8, "An Introduction to the C Shell", is based on a paper by William Joy describing the C shell command language interpreter. Also included in this chapter are a perforated quick reference card for use next to your terminal, a section describing special characters in the C Shell, and a glossary of terms.
- Chapter 9, "Communication Tutorial", teaches you how to send messages and files to users of both your system and other UNIX systems.

## Reference Information

Four appendices and a glossary of UNIX system terms are provided for quick reference at the end of this book.

- Appendix A, "Summary of the File System", illustrates how information is stored in the UNIX operating system.
- Appendix B, "Summary of UNIX System Commands", describes each UNIX system command discussed in the guide.
- Appendix C, "Summary of Shell Command Language", is a summary of the shell command language, notation, and programming constructs, as discussed in Chapter 6, "The Bourne Shell Tutorial".
- Appendix D, "Setting Up the Terminal", explains how to configure your terminal for use with the UNIX system.
- The Glossary defines terms used in this book pertaining to the UNIX system.

# Notation Conventions

The following notation conventions are used throughout this guide.

| | |
|---|---|
| **bold** | User input, such as commands, options and arguments to commands, variables, and the names of directories and files, appear in **bold.** |
| *italic* | Names of variables to which values must be assigned (such as *password*) appear in *italic.* |
| `typewriter font` | UNIX system output, such as prompt signs and responses to commands, appear in `typewriter font`. |
| <> | Input that does not appear on the screen when typed, such as passwords, tabs, or return, appear between angle brackets. |
| <**ctrl-***char*> | Control characters are shown between angle brackets because they do not appear on the screen when typed. To type a control character, hold down the control key while you type the character specified by *char*. For example, the notation **<ctrl-d>** means to hold down the control key while pressing the d key; the letter d will not appear on the screen. |
| [ ] | Command options and arguments that are optional, such as **[-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:<br><br>*command* [*arg1* \| *arg2*]<br><br>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.<br><br>Arrows on the screen (shown in examples in Chapter 5) represent the cursor. |
| *command(number)* | A command name followed by a number in parentheses refers to the part of a UNIX system reference manual that documents that command. |

(There are three reference manuals: the *IRIS-4D User's Reference Manual*, *IRIS-4D Programmer's Reference Manual*, and *IRIS-4D System Administrator's Reference Manual*.) For example, the notation **cat**(1) refers to the page in section 1 (of the *UNIX 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, different systems use different prompts. The Bourne shell, for example, uses $ as the shell prompt. Whichever symbol your system uses, keep in mind that a prompt is produced by the system; you are not meant to type it.

In all chapters, full and partial screens are used to display how your screen will look when you interact with the UNIX system. These examples show how to use the UNIX system editors, write short programs, and execute commands. 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 provided so you can experiment with the commands. The answers to all the exercises in a chapter are at the end of that chapter.

NOTE

The text in the *UNIX User's Guide* was prepared with the UNIX system text editors described in the guide and formatted with the Documenter's Workbench Software: **troff**, **tbl**, **pic**, and **mm** macros.

# What the UNIX System Does

The UNIX operating system is a set of programs that controls the computer. It acts as the link between you and the computer, providing tools to help you do your work. It is designed to provide an uncomplicated, efficient, and flexible computing environment. Specifically, the UNIX system offers the following:

- a general-purpose system for performing a 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 UNIX system interacts between users on a rotating basis so quickly that it appears to be interacting with all users simultaneously.
- a multi-tasking environment that enables you to execute more than one program simultaneously.

The organization of the UNIX system is based on four major components:

| | |
|---|---|
| the kernel | The kernel constitutes the nucleus of the operating system; it coordinates the functioning of the computer's internals (such as allocating system resources). The kernel works invisibly; you need never be aware of it while doing your work. |
| the file system | The file system provides a method of handling data that makes it easy to store and access information. |
| the shell | The shell 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 | Commands are programs that you request the computer to execute. Packages of programs are called tools. The UNIX system provides tools for jobs such as creating and changing text, writing programs and developing software tools, and exchanging information with others. |

# How the UNIX System Works

Each circle represents one of the main components of the UNIX 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 UNIX system, the file system. Figure 1-1 is a model of the UNIX system.

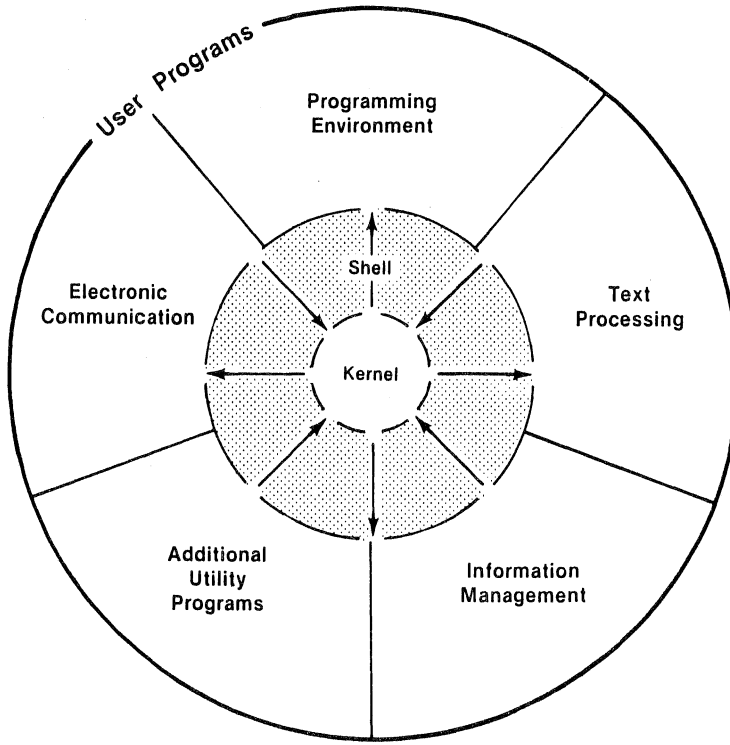



Figure 1-1: Model of the UNIX System

## The Kernel

The nucleus of the UNIX 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.

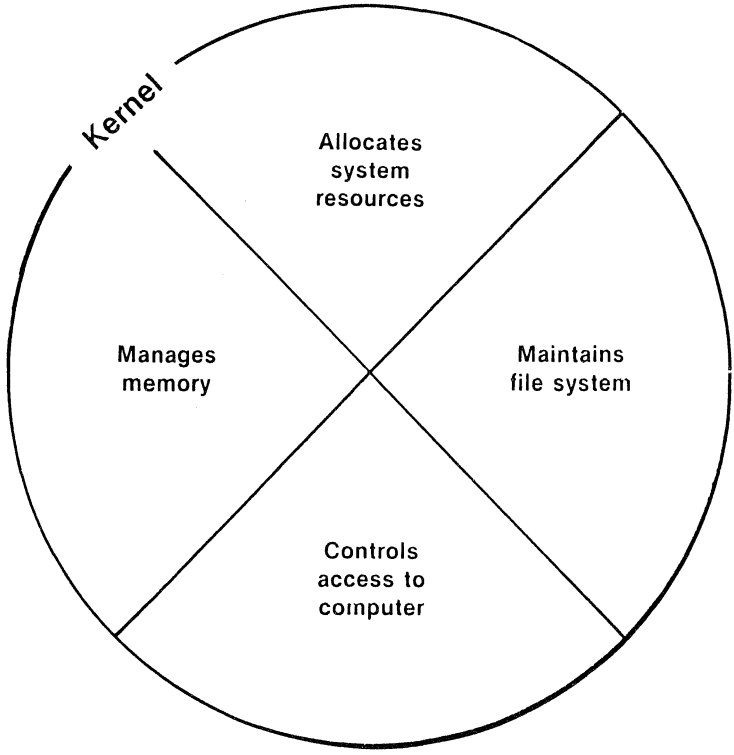



Figure 1-2: Functional View of the Kernel

# The File System

The file system is the cornerstone of the UNIX operating system. It provides a logical method of organizing, retrieving, and managing information. The structure of the file system is hierarchical; if you could see it, it might look like an organization chart or an inverted tree as shown in Figure 1-3.

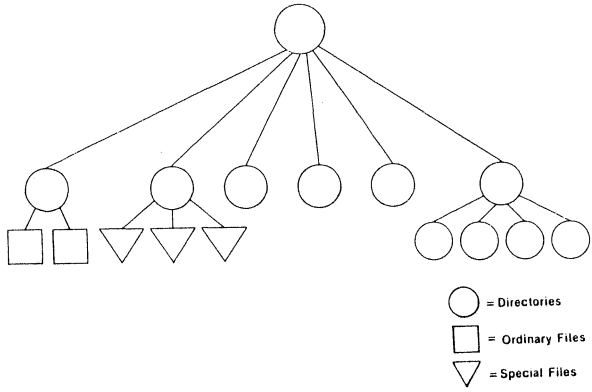



Figure 1-3: The Hierarchical Structure of the File System

The file is the basic unit of the UNIX system and it can be any one of three types: an ordinary file, a directory, or a special file. (See Chapter 3, "Using the UNIX File System.")

## Ordinary Files

An ordinary file is a collection of characters that is treated as a unit by the system. Ordinary files are used to store information you want to save. They may contain text for letters or reports, code for the programs you write, or commands to run your 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**, and so on. You can create directories, add or remove files from them, or remove directories themselves at any time.

All the directories that you create and own will be 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.

The UNIX system maintains several directories for its own use. These directories, which include **/unix** (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 UNIX 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 the UNIX system, however, all files are alike. This makes the UNIX 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 if you or a program you write needs to access a certain device, such as a printer, you specify the device just as you would another one of your files. In the UNIX 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.

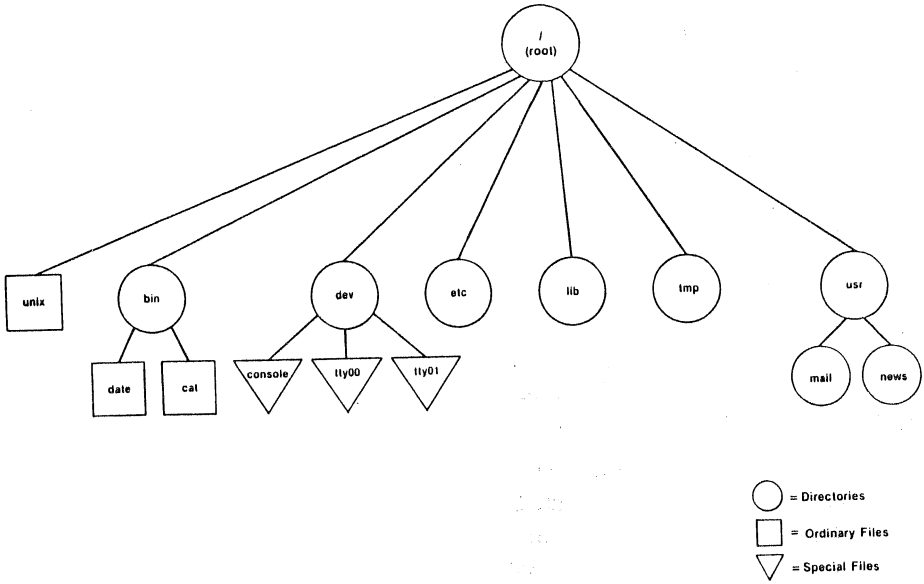



Figure 1-4: Example of a File System

In summary, the directories and files you create comprise the portion of the file system that you control. Other parts of the file system are provided and maintained by the operating system, such as **/bin, /dev, /etc, /lib, /tmp**, and **/usr**,

**/bin** contains many executable programs and utilities

**/dev** contains special files that represent peripheral devices such as the console, the 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. |

You will learn more about the file system in other chapters. Chapter 3 shows you how to organize a file system directory structure and how to access and manipulate files. Chapter 4 gives an overview of the UNIX system and describes the capabilities of the shells. 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. It reads the commands you enter and interprets them as requests to execute other programs, access files, or provide output.

There are two standard shells for the UNIX system. The Bourne shell, **sh**, and the C Shell, **csh**, are included in the Silicon Graphics, Inc. UNIX system package. Each of the shells has its own specific use but both shells can be used as interactive command interpreters and as programming languages.

**sch** is a subset of **sh** and includes the interactive commands you will most likely use to move between the shells. The primary purpose of **sch** is to translate command lines typed at a terminal into system actions. For more information on the shells see the *IRIS-4D User's Reference Manual.*

A shell is a powerful programming language, not unlike the C programming language, that provides conditional execution and control-flow features. The model of a UNIX system in Figure 1-1 shows the two-way flow of communication between you and the computer via the shell.

Chapter 7 is a tutorial that teaches you how to write simple Bourne shell programs, called shell scripts, and how to custom tailor your environment. Chapter 8 is a paper written by William Joy which describes the C shell and its various uses as a command language interpreter.

# 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 UNIX system, you have many standard programs and tools available to you. If you use the UNIX system to write programs and develop software, you can also draw on system calls, subroutines, and other tools. Of course, any programs you write will be at your disposal, too.

This book introduces you to many of the UNIX 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 *IRIS-4D User's Reference Manual.* For information on tools and routines, consult the *IRIS-4D Programmer's Reference Manual.*

The reference manuals may also be available online. (On-line documents are stored in your computer's file system.) You can summon pages from the on-line manuals by executing the command **man** (short for manual page). For details on how to use the **man** command, refer to the **man**(1) page in the *IRIS-4D User's Reference Manual.*

## What Commands Do

The outer circle of the UNIX 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 for creating and changing text, a spelling checker for locating spelling errors, and optional text formatters for producing 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, such as **mail**, enable you to transmit information to other users and to other UNIX systems. |
| programming environment | Several UNIX system programs establish a friendly programming environment by providing UNIX-to-programming language interfaces and by supplying numerous utility programs. |

| | |
|---|---|
| additional utilities | The system also offers capabilities for generating graphics and performing calculations. |

## How to Execute Commands

To make your requests comprehensible to the UNIX 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 UNIX system shell will not be able to interpret your request. Here is an example of the syntax of a UNIX system command line.

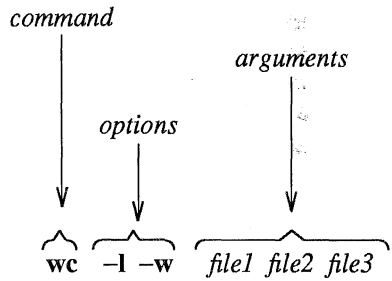
*command* *option(s)* *argument(s)*<**return**>

- 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)

On every UNIX system command line you must type at least two components: a command name and the return key. A command line may also contain either options or arguments, or both.

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

Some commands allow you to specify multiple options and/or 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 UNIX 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 **<return>**.

## How Commands Are Executed

Figure 1-5 shows the flow of control when the UNIX system executes a command.

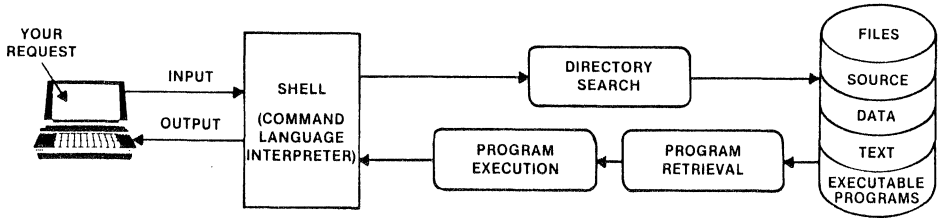



Figure 1-5: Execution of a UNIX System Command

To execute a command, enter a command line when a prompt (such as a % 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, along 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 printing another prompt.

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

# Getting Started With UNIX

This chapter acquaints you with the general rules and guidelines for working on the UNIX system. Specifically, it lists the required terminal settings, explains how to use the keyboard, obtain a login, log on and off the system, and enter simple commands.

To establish contact with the UNIX system, you need:

- a terminal
- a login name (a name by which the UNIX system identifies you as one of its authorized users)
- a password that verifies your identity
- instructions for dialing in and accessing the UNIX system if your terminal is not directly connected or hard-wired to the workstation

This chapter follows the notation conventions used throughout this guide. For a description of them, see the Preface.

## Required Terminal Settings

Regardless of the type of terminal you use, you must configure it properly to communicate with the UNIX system. If you have not set options before, you might feel more comfortable seeking help from someone who has.

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

The following is a list of configuration checks you should perform before trying to log in on the UNIX system.

1. Turn on the power.
2. Set the terminal to on-line 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 UNIX system.
4. If your terminal is not directly connected or 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 UNIX system will accommodate it by printing everything in uppercase letters.
6. Set the terminal to no parity.
7. Set the baud rate. This is the speed at which the computer communicates with the terminal, measured in characters per second. (For example, a terminal set at a baud rate of 4,800 sends and receives 480 characters per second.) Depending on the computer and the terminal, baud rates between 300 and 19,200 are available. Some computers may be capable of processing characters at higher speeds.

## 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 (such as the letters of the alphabet), each one is also associated with an ASCII code that is meaningful to the computer.

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.

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 UNIX system, you should be familiar with its typing conventions. The UNIX system requires that you enter commands in lowercase letters (unless the command includes an uppercase letter). Other conventions enable you to perform tasks, such as 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. Figure 2-1 lists the conventions based on special characters. Detailed explanations are provided on the next few pages.

| Key(s) | Meaning |
|---|---|
| % | System's command prompt (your cue to issue a command) |
| **<ctrl-u>** | Erase or kill an entire line |
| **<ctrl-c>** | Stop execution of a program or command |
| **<escape>** | 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 |
| **<return>**** | This ends a line of typing and puts the cursor on a new line. |
| **<ctrl-d>**† | Stop input to the system or log off |
| **<ctrl-h>** | Backspace for terminals without a backspace key |
| **<ctrl-i>** | Horizontal tab for terminals without a tab key |
| **<ctrl-s>** | Temporarily stops output from printing on the screen |
| **<ctrl-q>** | Makes the output resume printing on the screen after it has been stopped by the **<ctrl-s>** command |

* Nonprinting characters are shown in angle brackets (< >).

** The IRIS-4D "Return Key" is labeled "ENTER". Press the ENTER key whenever <return> is used in this manual.

† Characters preceded by a (**ctrl-**) are called control characters and are pronounced control-*letter*. To type a control character, hold down the control key and press the specified letter.

Figure 2-1: UNIX System Typing Conventions

## The Command Prompt

The standard UNIX system command prompt in the C shell in the (%) symbol. The Bourne shell prompt is the ($) symbol. When the prompt appears on your terminal screen, the UNIX system is waiting for instructions from you. The appropriate response to the prompt is to issue a command and press **<return>**.

Chapter 7 explains how to change the default prompt if you would prefer another character or character string as your command prompt.

## Correcting Typing Errors

There are two keys you can use to delete text so that you can correct typing errors. The @ (at) sign key kills the current line and the # (pound) sign key erases the last character typed. These keys are available by default to perform these functions. However, if you want to use other keys, you can reassign the erase and kill functions. (For instructions, see "Reassigning the Delete Functions" later in this section and "Setting Terminal Options" in Chapter 7.)

### Deleting the Current Line: the @ Sign

The @ sign key kills the current line. When you press it, an @ sign is added to the end of the line, and the cursor moves to the next line. The line containing the error is not erased from the screen but is ignored.

The @ sign key works only on the current line; be sure to press it before you press **<return>** 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 @ sign:

```
whooo@
who<return>
```

### Deleting the Last Characters Typed: the # Sign Key

The # (pound) sign key deletes the character(s) last typed on the current line. When you type a # sign, the cursor backs up over the last character and lets you retype it, thus effectively erasing 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 # signs. For example, in the following command line, two characters are deleted by typing two # signs.

```
dattw##e<return>
```

The UNIX system interprets this as the **date** command, typed correctly.

### The Backspace Key

Many people prefer to use the backspace key for the erase function instead of the # sign key. When you press **<backspace>**, the cursor backs up over your errors, erasing them as it goes. It does not print anything, unlike the # sign key, which prints a # sign on your screen between an error and a correction. When you have finished correcting an error with the backspace key, the line of text on the screen looks as though it was typed perfectly.

The # sign and backspace keys are equally effective at deleting characters, but using the backspace key gives you better visual information about what you are doing.


Some terminals may not recognize the # sign key as a delete character.

### 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 the **.profile** file in the Bourne shell or the **.login** file in the C shell. Instructions for making both temporary and permanent key reassignments, along with a description of **.profile**, are given in Chapter 7. See Chapter 8 for key reassignment instructions and a description of **.login**.

There are three points to keep in mind if you reassign the delete functions to non-default keys. First, the UNIX 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 # sign key to the backspace key, you will no longer be able to use the # sign key to erase characters. Neither will you have two keys that perform the same function.

Secondly, such reassignments are inherited by any other UNIX system program that allows you to perform the function you have reassigned. For example, the interactive text editor called **ed** (described in Chapter 6) 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 # sign key will no longer work.

Finally, keep in mind that any reassignments you have specified in your **.profile** or **.login** 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 # sign 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 **<return>** at the end of a line.

## Using Special Characters as Literal Characters

What happens if you want to use a special character as a unit of text? Since the UNIX system's 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, suppose you want to add the following sentence to a file:

Only one # appears on this sheet of music.

To prevent the UNIX system from interpreting the # sign as a request to delete a character, enter a \ in front of the # sign. 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 UNIX 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 UNIX system's work because the UNIX system 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, simply press **<break>** or **<delete>**. The UNIX system will stop the program and print a prompt on the screen, signalling that it has stopped the last command and is ready for your next command.

## 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 control key is used in combination with other characters to perform physical controlling actions on lines of type. 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 UNIX 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 control key and press the appropriate alphabetic key. Most control characters do not appear on the screen when typed but are shown between angle brackets (see "Notation Conventions" in the Preface).

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 **<ctrl-s>**; the printing will stop. When you are ready to read more output, type **<ctrl-q>** and the printing will resume.

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

In addition, the UNIX 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 **<ctrl-h>** key instead. You can also set tabs without a tab key by typing **<ctrl-i>** if your terminal is set properly. (Refer to the section in this chapter entitled "Possible Problems When Logging In" for information on how to set the tab key.)

Now that you have configured the terminal and inspected the keyboard, you must obtain a login name.

# Obtaining a Login Name

A login name is the name by which the UNIX system verifies that you are an authorized user 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 a UNIX system account through your local system administrator. There are a 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 will probably be determined by local practices. The system users may 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 with the UNIX System

Typically, you will be using either a workstation console that communicates directly with a workstation, or a terminal that communicates with a workstation over a telephone line.


This section describes a typical procedure for logging in, but may not apply to your system. There are many ways to log in on a UNIX 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 on your UNIX system from outside your computer installation site, see your system administrator.

Turn on your terminal. If it is directly connected, the `login:` prompt will immediately appear in the upper-lefthand corner of the screen.

If you are going to communicate with the workstation over a telephone line, you must now establish a connection. The following procedure is an example of a method you might use to do this. (For the procedure required by your system, see your system administrator.)

1. Dial the telephone number that connects you to the UNIX system. You will hear one of the following:
   - 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.
   - A high-pitched tone. This means that the system is accessible.
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. Replace the handset in the cradle.
3. After a few seconds, the `login:` prompt will appear in the upper-lefthand 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 UNIX system is trying to communicate with your terminal, but is using the wrong speed. Press **<break>** or **<return>**; this signals the system to try another speed. If the UNIX system does not display the `login:` prompt within a few seconds, press **<break>** or **<return>** again.

## Login Procedure

When the `login:` prompt appears, type your login name and press **<return>**. For example, if your login name is **starship**, your login line will look like this:

```
login: starship<return>
```

NOTE

Remember to type in lowercase letters. If you use uppercase from the time you log in, the UNIX system will expect and respond in uppercase exclusively until the next time you log in. It will accept and run many commands typed in uppercase, but will not allow you to edit files.

## Password

Next, the system prompts you for your password. Type your password and press **<return>**. For security reasons, the UNIX system does not print (or echo) your password on the screen.

If both your login name and password are acceptable to the UNIX system, the system may print the message of the day and/or current news items and then 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 will look similar to this:

```
login: starship<return>
password:
UNIX system news
%
```

If you make a typing mistake when logging in, the UNIX 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: ttarship<return>
password:
login incorrect
login:
```

The login procedure may also fail if the communication link between your terminal and the UNIX system has been dropped. If this happens, you must reestablish contact with the workstation (specifically, with the data switch that links your terminal to the workstation) before trying to log in again. 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 UNIX 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 depend on keeping your password private.

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

1. You establish contact; the UNIX system displays the `login:` prompt. Type your login name and press **<return>**.
2. The UNIX system prints the password prompt. Type your temporary password and press **<return>**.

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. For comparison purposes, an uppercase letter and its corresponding lowercase letter are equivalent.

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

NOTE

The UNIX 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 reenter your new password. Type your new password again.
7. If you do not reenter 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 reenter 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 summarizes this procedure (steps 1 through 6) for first-time UNIX system users.

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

## Possible Problems when Logging In

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

Some problems can be corrected simply 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:

| | |
|---|---|
| the 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. |
| the data phone set or modem | If your terminal is connected to the computer via telephone lines, verify that the baud rate and duplex settings are correctly specified. |
| the switches | Some terminals have several switches that must be set to be compatible with the UNIX system. If this is the case with the terminal you are using, make sure they are set properly. |

Refer to the section "Required Terminal Settings" in this chapter if you need information to verify the terminal configuration. If you need additional information about the keyboard, terminal, data phone, or modem, check the owner's manuals for the appropriate equipment.

Figure 2-2 presents a list of procedures you can follow to detect, diagnose, and correct some problems you may experience when logging in. If you need further help, contact your system administrator.

| Problem† | Possible Cause | Action/Remedy |
|---|---|---|
| Meaningless characters | UNIX system at wrong speed | Press <return> or <break> key |
| Input/output appears in uppercase letters | Terminal configuration includes uppercase setting | Log off and set character generation to lowercase |
| Input appears in uppercase, output in lowercase | Key labeled caps (or caps lock) is enabled | Press <caps> or <caps lock> to disable setting |
| Input is printed twice | Terminal is set to half-duplex mode | Change setting to full-duplex mode |
| Tab key does not work properly | Tabs are not set correctly | Type **stty –tabs**‡ |
| Communication link cannot be established although high-pitched tone is heard when dialing in | Terminal is set to local or off-line mode | Set terminal to on-line mode try logging in again |
| Communication link (terminal to UNIX system) is repeatedly dropped | Bad telephone line or bad communications port | Call system administrator |

* Numerous problems can occur if your terminal is not configured properly. To eliminate these possibilities before attempting to log in, perform the configuration checks listed under "Required Terminal Settings."

† Some problems may be specific to your terminal, data phone set, or modem. Check the owner's manual for the appropriate equipment if suggested actions do not remedy the problem.

‡ Typing **stty –tabs** corrects the tab setting for your current computing session. To ensure a correct tab setting for all sessions, add the line **stty –tabs** to your **.profile** or to your **.login** (see Chapter 7 or 8 respectively for details).

Figure 2-2: Troubleshooting Problems When Logging In*

## Simple Commands

When the prompt appears on your screen, the UNIX 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 **<return>**. The UNIX system accesses a program called **date**, executes it, and prints its results on the screen, as shown below.

```
% date<return>
Wed Oct 15 09:49:44 EDT 1986
%
```

As you can see, the **date** command prints the date and time, using the 24-hour clock.

Now type the **who** command and press **<return>**. Your screen will look something like this:

```
% who<return>
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.

## Logging Off

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

```
% <ctrl-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 UNIX system by typing **<ctrl-d>** before you turn off the terminal or hang up the telephone. If you do not, you may not be 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, see the "Special Commands" section of the **sh**(1) page in the *IRIS-4D User's Reference Manual.*)

# Preparing To Use The File System

To use the UNIX 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 ("How the File System is Structured" and "Your Place in the File System") offer a working perspective of the file system. The rest of the chapter introduces UNIX system commands that allow you to build your own directory structure, access and 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 UNIX system 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.

# How the File System is Structured

The file system is comprised 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 will review 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, such as 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 set of all the directories and files is 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, as shown in Figure 3-1.

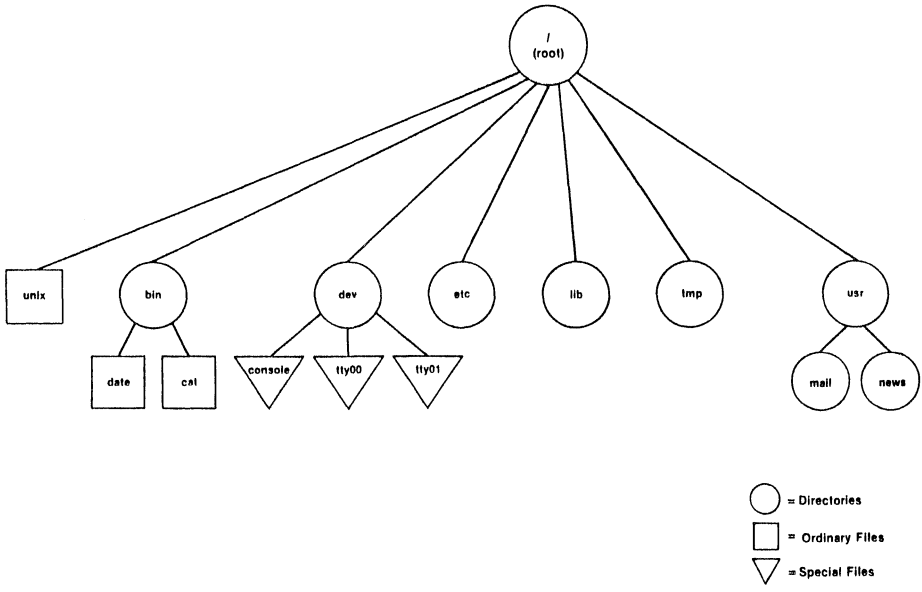



Figure 3-1: A Sample File System

# Your Place in the File System

Whenever you interact with the UNIX system, you do so from a location in its file system structure. The UNIX 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 UNIX 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 UNIX 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 UNIX 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 UNIX system directories discussed in Chapter 1.

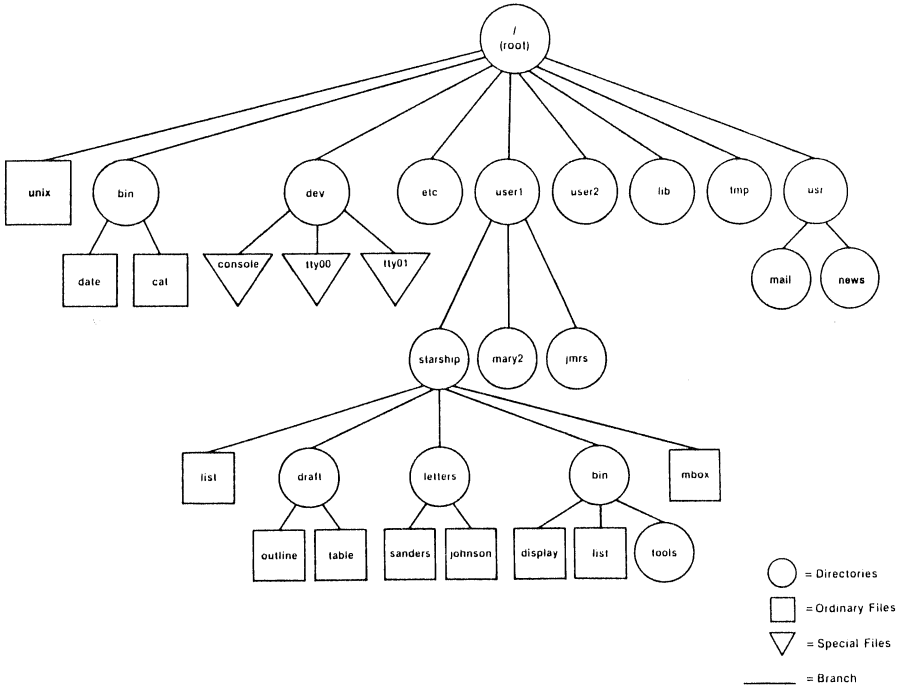



Figure 3-2: Directory of Home Directories

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 you can 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 UNIX 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 UNIX system will respond as follows:

```
% pwd<return>
/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 pathname `/user1/starship` tells you that the root directory (shown by the leading / in the line) contains the directory **user1** which, in turn, contains the directory **starship**. (All other slashes in the pathname 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 pathname. In the next few pages we will analyze and trace this pathname 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 especially helpful if you want to read or copy a file and the UNIX 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.

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

| **Command Recap**<br>**pwd** – print full name of working directory | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **pwd** | none | none |
| **Description:** | **pwd** prints the full pathname of the directory in which you are currently working. | |

Figure 3-3: Summary of the **pwd** Command

## Pathnames

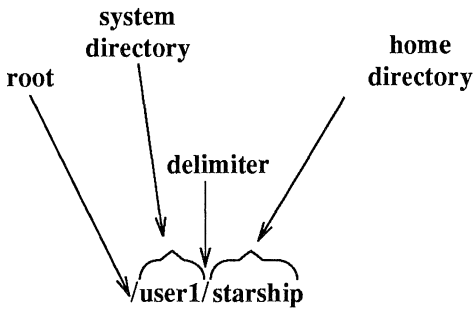
Every file and directory in the UNIX system is identified by a unique pathname. The pathname shows the location of the file or directory, and provides directions for reaching it. Knowing how to follow the directions given by a pathname is your key to moving around the file system successfully. The first step in learning about these directions is to learn about the two types of pathnames: full and relative.

## Full Pathnames

A full pathname (sometimes called an absolute pathname) gives directions that start in the root directory and lead you down through a unique sequence of directories to a particular directory or file. You can use a full pathname to reach any file or directory in the UNIX system in which you are working.

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

To understand how a full pathname 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 pathname of your working directory: `/user1/starship`. Analyze the elements of this pathname using the following diagram and key.




| | | |
|---|---|---|
| / (leading) | = | the slash that appears as the first character in the pathname 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 3-4 to trace the full path to **/user1/starship.**

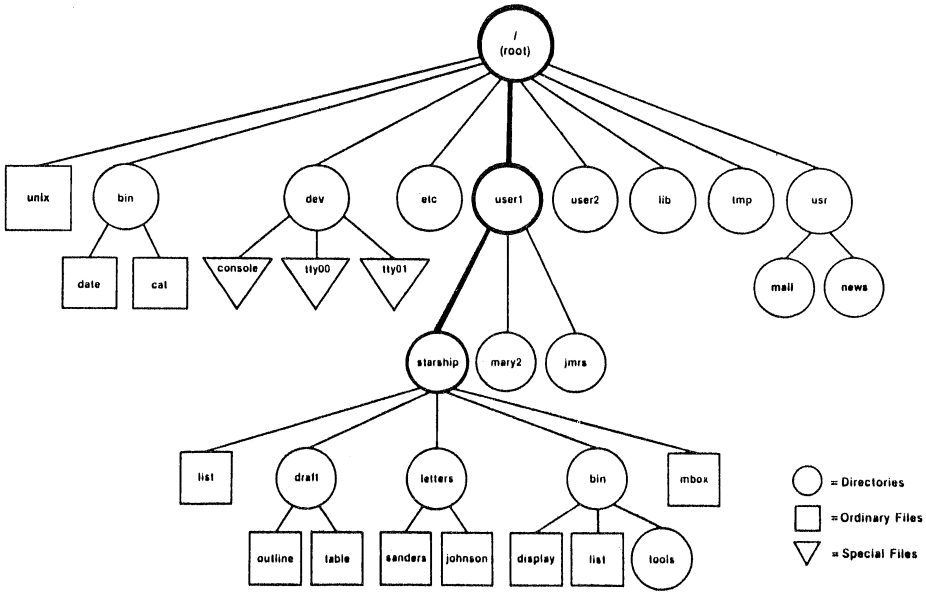



Figure 3-4: Full Pathname of the **/user1/starship** Directory

## Relative Pathnames

A relative pathname 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 pathname begins with one of the following: 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 .. (dot dot) is called the parent directory of . (your current directory).

For example, say 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 pathname to any of these is simply its name, such as **draft** or **mbox**. Figure 3-5 traces the relative path from **starship** to **draft**.

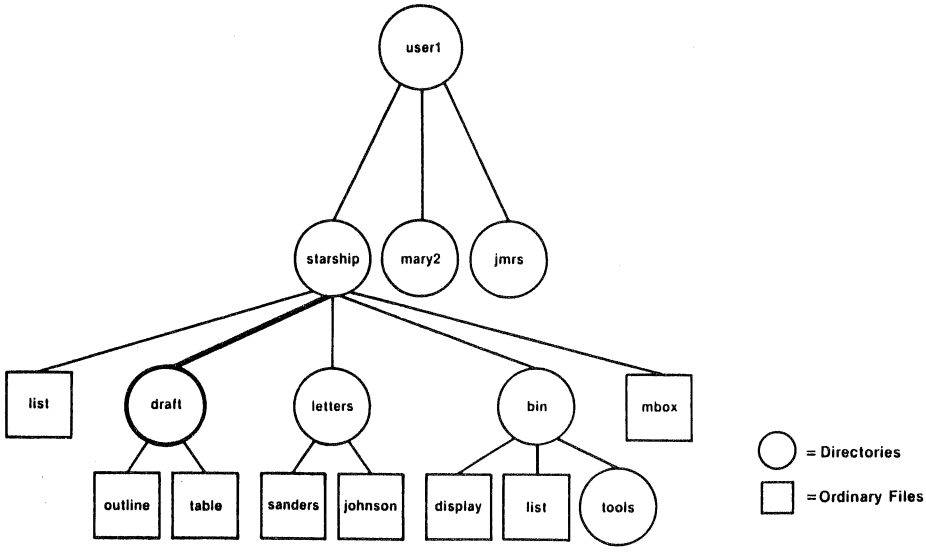



Figure 3-5: Relative Pathname of the **draft** Directory

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

Figure 3-6 traces this relative path. Notice that the slash in this pathname separates the directory named **draft** from the file named **outline**. Here, the slash is a delimiter showing that **outline** is subordinate to **draft**; that is, **outline** is a child of **draft**.

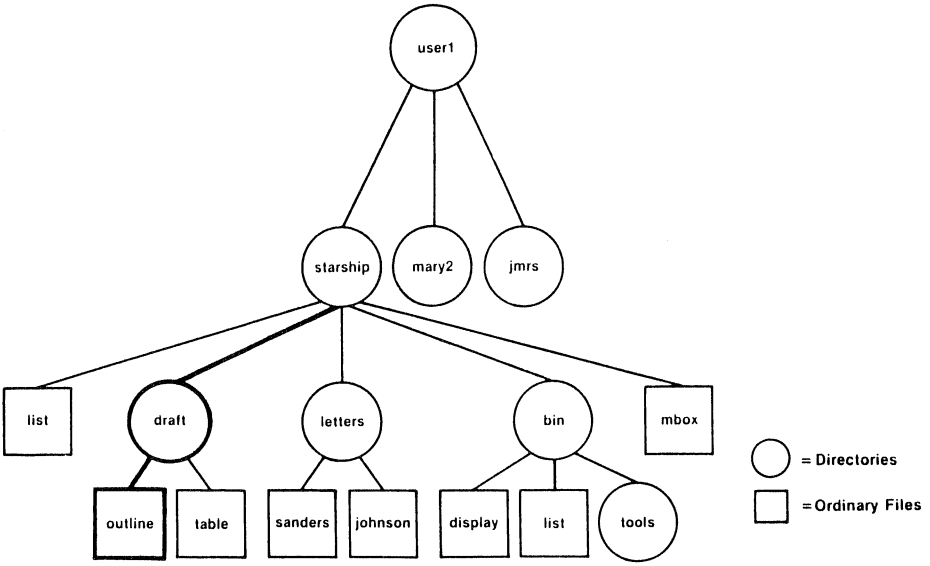



Figure 3-6: Relative Pathname from **starship** to **outline**

So far, the discussion of relative pathnames 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 and 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 pathname to **starship**, and .. is the pathname to **starship**'s parent directory, **user1**.

From **draft**, you can also trace a path to the file **sanders** by using the pathname **../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 pathname in place of a relative one.

Figure 3-7 shows some examples of full and relative pathnames.

| Path Name | Meaning |
|---|---|
| / | full pathname of the root directory |
| **/bin** | full pathname of the **bin** directory (contains most executable programs and utilities) |
| **/user1/starship/bin/tools** | full pathname 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 pathname to the file or directory **tools** in the directory **bin**<br><br>If the current directory is /, then the UNIX 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 pathname of a file or directory **tools** in the current directory. |

Figure 3-7: Example Pathnames

You may need some practice before you can use pathnames such as these to move around the file system with confidence.

## 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 one to 14 characters long.

- All characters other than / are legal.
- Some characters are best avoided, such as a 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 UNIX 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 UNIX system commands that enable you to examine the file system.

# Organizing a Directory

This section introduces four UNIX 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 |

These commands can be used with either full or relative pathnames. Two of the commands, **ls** and **cd**, can also be used without a pathname. Each command is described more fully in the four sections that follow.

## Creating Directories: the 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 pertaining to one subject together in a directory, you will know where to find them later.

To create a directory, use the command **mkdir** (short for make directory). Simply enter the command name, followed by the name you are giving your new directory or file. 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 <return>
%
```

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, such as **letters** and **bin**, in the same way.

```
% mkdir letters<return>
% mkdir bin<return>
%
```

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<return>
%
```

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."

Figure 3-8 summarizes the syntax and capabilities of the **mkdir** command.

| **Command Recap**<br>**mkdir** – make a new directory | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **mkdir** | none | *directoryname(s)* |
| **Description:** | **mkdir** creates a new directory (subdirectory). | |
| **Remarks:** | The system returns a prompt (% by default) if the directory is successfully created. | |

Figure 3-8: Summary of the **mkdir** Command

## Listing the Contents of a Directory: the ls Command

All directories in the file system have information about the files and directories they contain, such as 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 3-1) once again.

Say you are logged in to the UNIX system and you run the **pwd** command. The system responds with the pathname **/user1/starship**. To display the names of files and directories in this current directory, you then type **ls** and press **<return>**. After this sequence, your terminal will read:

```
% pwd<return>
%/user1/starship
% ls<return>
bin
draft
letters
list
mbox
%
```

As you can see, 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 have been printed first.)

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

**ls** *pathname***<return>**

The directory name can be either the full or relative pathname 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 **<return>**. Your screen will look like this:

```
% ls draft<return>
outline
table
%
```

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

You can also use a relative pathname 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 pathname from **starship** to **user1**:

```
% ls ..<return>
jmrs
mary2
starship
%
```

You can get the same results by using the full pathname from root to **user1**. If you type **ls /user1** and press 8**<return>**, the system will respond by printing the same list.

Similarly, you can list the contents of any system directory that you have permission to access by executing the **ls** command with a full or relative pathname.

The **ls** command is useful if you have a long list of files and you are trying to determine whether 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<return>
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. These commands 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 UNIX system. Refer to the **ls**(1) page in the *IRIS-4D User's Reference Manual* for details about other options.

### Listing All Names in a File

Some important file names in your home directory, such as **.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 ., type **ls –a** and press **<return>**.

```
% ls -a<return>
.
..
.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 *) and directories (with a /). Thus, you can list all files in your working directory **starship** by executing the command line shown here:

```
% ls -CF<return>
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, say you run the **ls -l** command while in the **starship** directory.

```
% ls -l<return>
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 users 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 is given next. Finally, the last column shows the name of the directory or file.

Figure 3-9 identifies each column in the rows of output from the **ls –l** command.

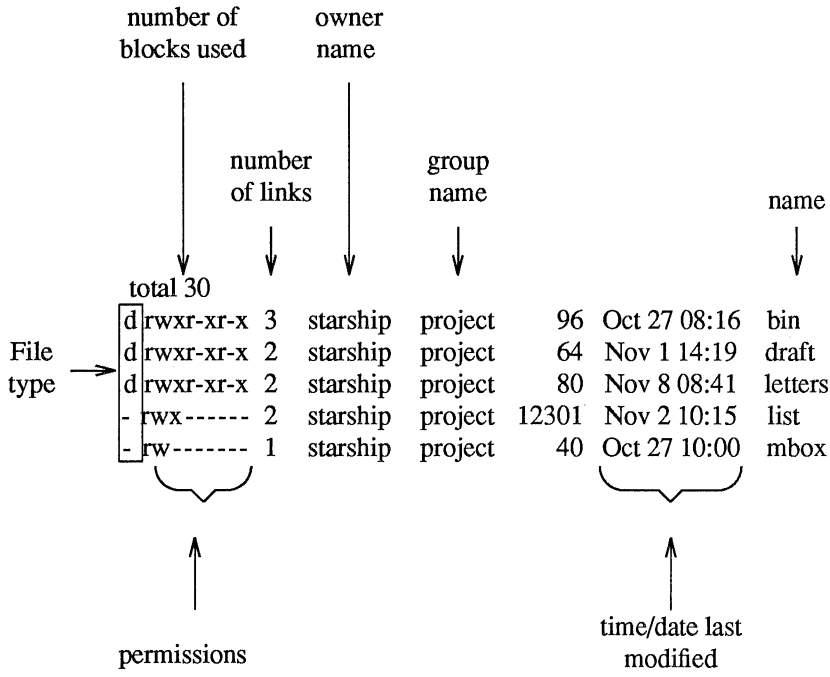



Figure 3-9: Description of Output Produced by the **ls –l** Command

Figure 3-10 summarizes the syntax and capabilities of the **ls** command and two available options.

| Command Recap<br>**ls** – list contents of a directory | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **ls** | **–a, –l,** and others* | *directoryname(s)* |
| **Description:** | **ls** lists the names of the files and subdirectories in the specified directories. If no directory name is given as an argument, the contents of your working directory are listed. | |
| **Options:** | **–a** Lists all entries, including those beginning with . (dot).<br>**–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 *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-10: Summary of the **ls** Command

# Changing Your Current Directory: the cd Command

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

**cd** *pathname_of_newdirectory*<**return**>

Any valid pathname (full or relative) can be used as an argument to the **cd** command. If you do not specify a pathname, the command will move 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 <**return**>. (Here **draft** is the relative pathname to the desired directory.) When you get a prompt, verify your new location by typing **pwd** and pressing <**return**>. Your terminal screen will look like this:

```
% cd draft<return>
% pwd<return>
/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. (Chapter 5 covers the **vi** editor commands and Chapter 6 covers the **ed** editor commands.)

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 pathname for it. For example, to **cat** the **sanders** file in the **letters** directory (**/user1/starship/letters**) while you are in the **draft** directory (**/user1/starship/draft**), specify the full pathname of **sanders** on the command line.

**cat /user1/starship/letters/sanders<return>**

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

**cd /user1/starship/letters<return>**

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

Figure 3-11 summarizes the syntax and capabilities of the **cd** command.

| **Command Recap**<br>**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 pathname in place of a simple directory name. | |

Figure 3-11: Summary of the **cd** Command

## Removing Directories: the 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)*<**return**>

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 (that is, is not empty), the **rmdir** command prints the message *directoryname* `not empty`. You must remove all subdirectories and files; only then will the command succeed.

For example, say 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 follows:

```
% rmdir memos<return>
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** itself 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<return>**

If **memos** is empty, the command will remove it and return a prompt.

Figure 3-12 summarizes the syntax and capabilities of the **rmdir** command.

| **Command Recap**<br>**rmdir** – remove a directory | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **rmdir** | none | *directoryname(s)* |
| **Description:** | **rmdir** removes specified directories if they do not contain files and/or subdirectories. | |
| **Remarks:** | If the directory is empty, it is removed and the system returns a prompt. If the directory contains files or subdirectories, the command returns the message, `rmdir:` *directoryname* `not empty.` | |

Figure 3-12: Summary of the **rmdir** Command

# Accessing and Manipulating Files

This section introduces several UNIX 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 UNIX system commands that are necessary for accessing and using the files in the directory structure. Figure 3-13 lists these commands.

| Command | Function |
|---|---|
| **cat** | prints the contents of a specified file on a terminal |
| **pg** | prints the contents of a specified file on a terminal in chunks or pages |
| **pr** | prints 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 file |
| **mv** | moves and renames a file |
| **rm** | removes a file |
| **wc** | reports the number of lines, words, and characters in a file |
| **chmod** | changes permission modes for a file (or a directory) |

Figure 3-13: Basic Commands for Using Files

Each command is discussed in detail and summarized in a table that you can easily reference later. These tables 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 UNIX system provides three commands for displaying and printing the contents of a file or files: **cat, pg,** and **pr**. The **cat** command (short for concatenate) outputs the contents of the file(s) specified. This output is displayed on your terminal screen unless you tell **cat** to direct it to another file or a new command.

The **pg** command is particularly 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 terminal or, if you so request, directs the formatted output to a printer (see 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 a file or files. For example, say 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:

```
% cat johnson<return>
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, simply 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**, type:

```
% cat johnson sanders<return>
```

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

```
% cat johnson sanders<return>
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, see the sections in Chapter 7 that discuss input and output redirection.

Figure 3-14 summarizes the syntax and capabilities of the **cat** command.

| Command Recap<br>**cat** – concatenate and print a file's contents | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **cat** | available* | *filename(s)* |
| **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 are displayed on the terminal screen; otherwise, the message `cat: cannot open` *filename* appears on the screen.<br>To display the contents of a directory, use the **ls** command. | |

* See the **cat**(1) page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-14: Summary of the **cat** Command

### Paging Through the Contents of a File: the pg Command

The command **pg** (short for page) allows you to examine the contents of a file or 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 the command to continue displaying the file's contents a page at a time, and a request that the command search through the file(s) to locate a specific character pattern. Figure 3-15 summarizes some of the available instructions.

| Command* | Function |
|---|---|
| **h** | help; display list of available **pg**† commands |
| **q** or **Q** | quit **pg** perusal mode |
| **<return>** | display next page of text |
| **l** | display next line of text |
| **d** or **<ctrl-d>** | display additional half page of text |
| **.** or **<ctrl-l>** | redisplay current page of text |
| **f** | skip next page of text and display following one |
| **n** | begin displaying next file you specified on command line |
| **p** | display previous file specified on command line |
| **%** | display last page of text in file currently displayed |
| */pattern* | search forward in file for specified character pattern |
| **?***pattern* | search backward in file for specified character pattern |

* 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 *IRIS-4D User's Reference Manual* for a detailed explanation of all available **pg** commands.

Figure 3-15: Summary of Commands to Use with **pg**

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 23 lines long, its page size will be the number of lines in the file plus one (for the colon).

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

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

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

**pg outline<return>**

The first page of the file will appear on the screen. Because the file has more lines in it than can be displayed 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** will print the next page of the file.

The following screen summarizes our discussion of the **pg** command this far.

```
% pg outline<return>
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
:<return>
```

After you press the return key, **pg** will resume 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 containing 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 will appear 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 (see the **pg**(1) page in the *IRIS-4D 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** or **.login** files. (For more information about TERM, see Chapter 7 for the Bourne shell or Chapter 8 for the C shell; for instructions on setting the TERM variable, see Appendix D.)

Figure 3-16 summarizes the syntax and capabilities of the **pg** command.

| **Command Recap**<br>**pg** – display a file's contents in chunks or pages | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **pg** | available* | *filename(s)* |
| **Description:** | The **pg** command displays the contents of the specified file(s) 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** perusal 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 been displayed. | |

* See the **pg**(1) page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-16: Summary of the **pg** Command

### Print Partially Formatted Contents of a File: the 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 terminal screen.

You have the option of requesting that the command print its output on another device, such as 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 sections on 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 formatted file is followed by five blank lines. (Complete sets of text-formatting tools are available on UNIX systems equipped with the Documenter's Workbench Software.)

The **pr** command is often used together 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 format and print the contents of a file on your terminal. For example, to review the contents of the file **johnson** in the sample file system, type:

**pr johnson<return>**

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

```
% pr johnson<return>


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 ellipses after the last line in the file represent the remaining lines (all blank in this case) that **pr** formatted into the output (so that each page contains 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 formatted file will be printed 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 **<ctrl-q>** to resume printing.

Figure 3-17 summarizes the syntax and capabilities of the **pr** command.

**Command Recap**

**pr** – print formatted contents of a file

| *command* | *options* | *arguments* |
|---|---|---|
| **pr** | available* | *filename(s)* |

**Description:** The **pr** command produces a formatted copy of a file(s) on your terminal 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 are formatted and displayed; if not, the message `pr: can't open` *filename* is printed.

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 **<ctrl-s>** and **<ctrl-q>**, respectively.

* See the **pr**(1) page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-17: Summary of the **pr** Command

## Requesting a Paper Copy: the 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. However, if you are using a video display terminal, you must send a request for a paper copy of a file to a printer. The command **lp** (short for line printer) allows you to do this.

To execute **lp**, follow this format:

**lp** *filename*<**return**>

For example, to print the file **johnson** on a line printer, type the following command line:

**lp johnson<return>**

The system responds with the name (or type) of the printer on which the file will be printed, and an identification (ID) number for your request.

```
% lp johnson<return>
request id is laser-6885  (1 file)
%
```

The system response shows that your job is to be printed on a laser printer (this system's default type of printer), has a request ID number of 6885, and includes one file.

The **–d***dest* (short for destination) option on the command line causes your file to be printed on another available device that you specify in the *dest* argument. The **–m** option causes mail to be sent to you stating the job has been completed.

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

**cancel laser-6885<return>**

To check the status of a line printer job that it is in progress, or to get its request ID number, execute the **lpstat** command. This command also provides a complete listing of every printer available on your system. Which printers are available to you depends on your UNIX system facility. Ask your system administrator for the names of available line printers, or type the following command line:

**lpstat –v<return>**

Figure 3-18 summarizes the syntax and capabilities of the **lp** command.

| **Command Recap** | | |
|---|---|---|
| **lp** – request paper copy of file from a line printer | | |
| *command* | *options* | *arguments* |
| **lp** | **–d, –m,** and others* | *file(s)* |
| **Description:** | The **lp** command requests that specified files be printed by a line printer, thus providing paper copies of the contents. | |
| **Options:** | **–d***dest* Allows you to choose *dest* as the printer or type of printer to produce the paper copy. If you do not use this option, the **lp** program specifies the printer for you.<br>**–m** Sends a message to you via **mail** after the printing is complete. | |
| **Remarks:** | You can cancel a request to the line printer by typing **cancel** and the request **ID** furnished to you by the system when the request was acknowledged.<br>Check with your system administrator for information on additional and/or different commands for printers that may be available at your location. | |

* See the **lp**(1) page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-18: Summary of the **lp** Command

## Making a Duplicate: the cp Command

When using the UNIX system, you may want to make a copy of a file. For example, you might want to 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, simply 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 summarizes these activities.

```
% cp outline new.outline<return>
% ls<return>
new.outline
outline
table
%
```

The UNIX 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 output on your screen:

```
% cp outline outline<return>
cp: outline and outline are identical
% ls<return>
outline
table
%
```

The UNIX 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<return>** (full pathname specified)
- **cp outline ../letters/outline<return>** (relative pathname specified)

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<return>** (full pathname specified)
- **cp outline ../letters<return>** (relative pathname specified)

In any of these four cases, **cp** will make a copy of the **outline** file in the **letters** directory and call it **outline**, too.

Of course, 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 the following command lines:

- **cp outline /user1/starship/letters/outline.vers2<return>** (full pathname)

- **cp outline ../letters/outline.vers2<return>** (relative pathname)

When assigning new names, keep in mind the conventions for naming directories and files described in "Naming Directories and Files" in this chapter.

Figure 3-19 summarizes the syntax and capabilities of the **cp** command.

| **Command Recap**<br>**cp** – make a copy of a file | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **cp** | none | *file1 file2*<br>*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.<br>You cannot copy directories with the **cp** command. | |

Figure 3-19: Summary of the **cp** Command

## Moving and Renaming a File: the 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*<**return**>

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 pathnames.

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**, simply type **mv table new.table** and press the return key. If the command executes successfully, you will 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 as follows:

```
% mv table new.table<return>
% ls<return>
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*<**return**>

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

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

**mv table /user1/starship/letters<return>**

**mv table /user1/starship/letters/table<return>**

**mv table ../letters<return>**

**mv table ../letters/table<return>**

**mv /user1/starship/draft/table /user1/starship/letters/table<return>**

Now suppose 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<return>**

**mv table ../letters/table2<return>**

**mv /user1/starship/draft/table2 /user1/starship/letters/table2<return>**

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

Figure 3-20 summarizes the syntax and capabilities of the **mv** command.

| Command Recap<br>**mv** – move or rename files | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **mv** | none | *file1 file2*<br>*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. | |

Figure 3-20: Summary of the **mv** Command

## Removing a File: the 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)*<**return**>

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*<**return**>

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. Since **ls** lists the files in your directory, you'll immediately be able to see whether or not **rm** has executed successfully.

For example, say 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 will be empty. Verify this by running the **ls** command.

```
% rm outline table <return>
% ls
%
```

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

Figure 3-21 summarizes the syntax and capabilities of the **rm** command.

| **Command Recap**<br>**rm** – remove a file | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **rm** | available* | *file(s)* |
| **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 *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-21: Summary of the **rm** Command

## Counting in a File: the wc Command

The command **wc** (short for word count) reports the number of lines, words, and characters there are in the file(s) 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<return>**

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<return>
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*<**return**>

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* are displayed 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<return>
    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***<return>** (line count)
**wc –w** *file1***<return>** (word count)
**wc –c** *file1***<return>** (character count)

For example, if you use the **–l** option, the system reports only the number of lines in **sanders.**

```
% wc –l sanders<return>
      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.

Figure 3-22 summarizes the syntax and capabilities of the **wc** command.

| **Command Recap**<br>**wc** – count lines, words, and characters in a file | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **wc** | **–l, –w, –c** | *file(s)* |
| **Description:** | **wc** counts lines, words, and characters in the specified file(s), 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)<br>**–w** counts the number of words in the specified file(s)<br>**–c** counts the number of characters in the specified file(s) | |
| **Remarks:** | When a file name is specified in the command line, it is printed with the count(s) requested. | |

Figure 3-22: Summary of the **wc** Command

## Protecting Your Files: the chmod Command

The command **chmod** (short for change mode) allows you to decide who can read, write, and use your files and who cannot. Because the UNIX operating system is a multi-user system, you usually do not work alone in the file system. System users can follow pathnames 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, to 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 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** you, the owner of your files and directories (**u** is short for user)
- **g** 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 UNIX system account)
- **o** 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 Chapter 7 for details on how to modify your environment in the Bourne shell, and Chapter 8 on how to modify your environment in the C shell). 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.

### How to Determine 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:

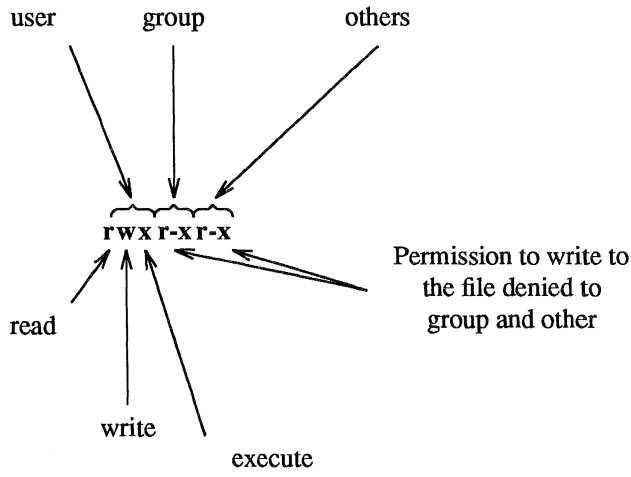
```
% ls -l<return>
total 35
-rwxr-xr-x   1 starship      project        9346  Nov 1  08:06  display
-rw-r--r--   1 starship      project        6428  Dec 2  10:24  list
drwx--x--x   2 starship      project          32  Nov 8  15:32  tools
%
```

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 (for example, a `-` (dash) symbolizes a regular file and a directory), the other nine characters that set the permissions comprise 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 `r`, `w`, and `x` permissions and members of the group and other system users have `r` and `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 *IRIS-4D 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`.

### How to Change Existing Permissions

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

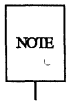
**chmod** *who+permission file(s)*<**return**>

or

**chmod** *who=permission file(s)*<**return**>

The following list defines each component of this command line.

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




The **chmod** command will not work if you type a space(s) between *who*, the instruction that gives (+) or denies (–) permission, and the *permission*.

The following examples show a few possible 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<return>**

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<return>
% ls -l<return>
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
%
```

As you can see, 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<return>**

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

```
% chmod go-r display<return>
% ls -l<return>
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
%
```

### A Note on Permissions and Directories

You can use the **chmod** command to grant or deny permission for directories as well as files. Simply 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, say 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.

### An Alternative Method

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 *IRIS-4D User's Reference Manual.*

Figure 3-23 summarizes the syntax and capabilities of the **chmod** command.

| Command Recap<br>**chmod** – change permission modes for files (and directories) | | |
|---|---|---|
| *command* | *instruction* | *arguments* |
| **chmod** | **who + – permission** | *filename(s)*<br>*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. | |

Figure 3-23: Summary of the **chmod** Command

# 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 just that.

**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 *IRIS-4D User's Reference Manual.*

## Identifying Differences: the 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*<**return**>

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 UNIX 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, say 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 will identify 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 the **johnson** file will be identical to the **mcdonough** file. Remember, the **diff** command identifies differences between specified files. If you want to make an identical copy of a file, use the **cp** command.

Figure 3-24 summarizes the **diff** command.

**Command Recap**

**diff** – finds differences between two files

| *command* | *options* | *arguments* |
|---|---|---|
| **diff** | available* | *file1 file2* |

| | |
|---|---|
| **Description:** | The **diff** command reports what 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 *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-24: Summary of the **diff** Command

## Searching a File for a Pattern: the grep Command

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

The basic format for the command line is:

**grep** *pattern file(s)*<return>

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

**grep automation johnson<return>**

The system responds:

```
% grep automation johnson<return>
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 UNIX system, (such as **%**, **|**, *****, **?**, and so on), the entire pattern must be enclosed in single quotes. (See the Chapter 7 section "Metacharacters".) For example, say 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<return>
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<return>
johnson:and office automation software.
%
```

This tells you that the pattern `office automation` is found once in the **johnson** file. In addition to the **grep** command, the UNIX 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 *IRIS-4D User's Reference Manual* for further information about these commands.

Figure 3-25 summarizes the syntax and capabilities of the **grep** command.

| Command Recap<br>**grep** – searches a file for a pattern | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **grep** | available* | *pattern file(s)* |
| **Description:** | The **grep** command searches through specified file(s) for lines containing a pattern and then prints the lines on which it finds the pattern. If 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 single quotes on the command line. | |

* See the **grep**(1) page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-25: Summary of the **grep** Command

## Sorting and Merging Files: the sort Command

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

**sort** *file(s)*<**return**>

This command causes lines in the specified files to be sorted and merged in the following order.

- Lines beginning with numbers are sorted by digit and listed before lines beginning with letters.

- Lines beginning with uppercase letters are listed before lines beginning with lowercase letters.
- Lines beginning with symbols such as * or @, are sorted on the basis of the symbol's ASCII representation.

For example, let's say 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.

```
% cat group1<return>
Smith, Allyn
Jones, Barbara
Cook, Karen
Moore, Peter
Wolf, Robert
% cat group2<return>
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 will be printed on the terminal screen unless you specify otherwise.

```
% sort group1 group2<return>
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 *IRIS-4D User's Reference Manual* for a full description of available options.

Figure 3-26 summarizes the syntax and capabilities of the **sort** command.

| Command Recap<br>**sort** – sorts and merges files | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **sort** | available* | *file(s)* |
| **Description:** | The **sort** command sorts and merges lines from a file or files you specify and displays its output on your terminal 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 *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-26: Summary of the **sort** Command

# Summary

This chapter described the structure of the file system and presented ways to use and to navigate through the file system by using UNIX system commands. The next chapter gives you an overview of a variety of UNIX system capabilities: text editing, using the shell as a command language, communicating electronically with other system users, and programming and developing software.

# Tutorial Overview

This chapter serves as a transition between the overviews that comprise the first three chapters and the tutorials in the following five chapters. Specifically, it provides an overview of the subjects covered in these tutorials: Text Editing, Working in the Shell, and Communicating Electronically. Text editing is covered in Chapter 5, "Screen Editor Tutorial," and Chapter 6, "Line Editor Tutorial". How to work and program in the Bourne shell is taught in Chapter 7, "Shell Tutorial"; working and programming in the C Shell is discussed in Chapter 8, "The C Shell"; and electronic communication is covered in Chapter 9, "Communication Tutorial".

# Text Editing

Using the file system is a way of life in a UNIX system 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 UNIX system: the screen editor, **vi**, (short for visual editor) and the line editor, **ed**. For detailed information about **vi** and **ed**, see 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 UNIX system text editors, like the UNIX 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. All the commands introduced up to this point belong to the shell's command set. 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.

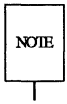
## How Does a Text Editor Work?

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 that 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 also temporary. 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 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 remind you to write your file if you attempt to end an editing session without doing so.




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 **<return>**. 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 line; others do not.

## Modes of Operation

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.

## Screen Editor

The screen editor, accessed by the **vi** command, is a display-oriented, interactive software tool. It allows you to view the file you are editing a page at a time. This editor works most efficiently when used on a video display terminal operating at 1,200 baud or higher.

For the most part, 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 and then making 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.

Furthermore, 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. 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 demand it places on the computer's processing time. Every time you make a change **vi** must update the screen. Despite this drawback, **vi** is considered the most useful editing tool. Chapter 6, "Screen Editor Tutorial," contains a **vi** quick reference card to tear out and use next to your workstation or terminal. Chapter 6 also includes detailed instructions on how to use the **vi** editor.

## 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.

The line editor (**ed**) works well on video display terminals and paper printing terminals. It will also work if you are using a slow-speed telephone line. (The visual editor, **vi**, can be used only on video display terminals.) See the appendix following Chapter 6 for a summary of line editor commands.

# The Shell

Every time you log in to the UNIX system you start communicating with the shell you choose to work with, and continue to do so until you log off the system. However, while you are using one of the text editors, 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 UNIX 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 to enhance your ability to use system features. In addition to using the shell 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 Bourne shell as a command interpreter and a programming language, refer to Chapter 7. For information on using the C shell, see Chapter 8.

## Customizing Your Computing Environment

This section deals with another control provided by each of the shells: your environment. When you log in to the UNIX system, the shell you choose automatically sets up a computing environment for you. The default environment set up by the Bourne shell includes these variables (the C shell variables are always lowercase):

| | |
|---|---|
| **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 for the duration of your current computing session, specify your changes in a command line (depending on the shell you are using, see Chapters 7 or 8 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** in the Bourne shell and must be located in your home directory. (In the C shell, **.login** sets up your environment when you log in and **.cshrc** sets up new environments as you open subsequent windows or new shells.)

The **.profile** and **.login** files typically perform some or all of the following tasks: check for mail; set data parameters, terminal settings, and tab stops; assign a character or character string as your login prompt; and assign the erase and kill functions to keys. You can define as few or as many tasks as you want in your **.profile** or **.login**. You can also change parts of it at any time. For instructions on modifying a **.profile**, see "Modifying Your Login Environment" in Chapter 7. To modify your **.login**, see "Shell Startup and Termination" in Chapter 8.

If you are using the Bourne shell, 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 will appear on your screen. If you do not have a **.profile** you can create one with a text editor, such as **ed** or **vi**. (See "Modifying Your Login Environment" in Chapter 7 for instructions.) Follow the same procedure using the **.login** file if you are using the C shell.

## Programming in the Shell

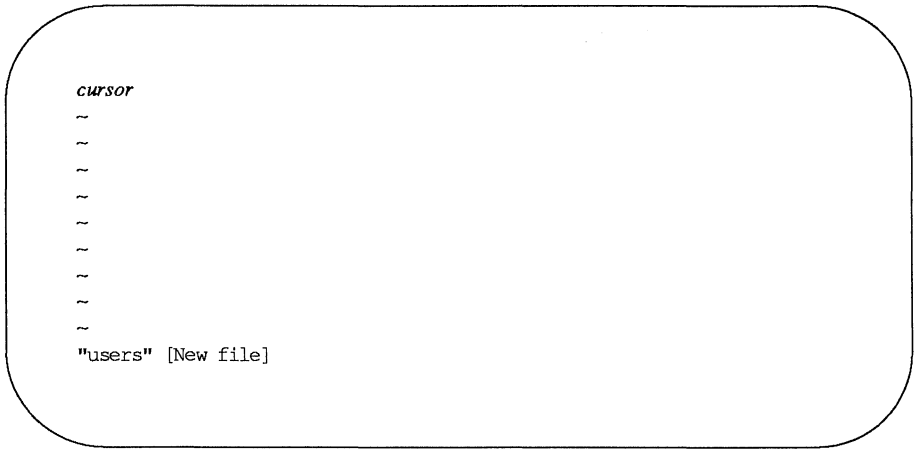
The shell is not only the command language interpreter; it is also a command level programming language. This means that instead of always using the shell strictly as a liaison between you and the computer, you can also program it to repeat sequences of instructions automatically. To do this, you must 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 simply 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 provides 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 UNIX system programs (the **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 5) to create your script, you can follow this procedure. First, create the file **users** with the editor by typing

**vi users<return>**

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




Enter the three UNIX 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<return>**

Now try running your new command. The following screen shows the kind of output you will get.

```
% users<return>
Sat Mar 1 16:40:12 EST 1986
      4
%
```

The output tells you that four users were logged in on the system when you typed the command at 16:40 on Saturday, March 1, 1986.

For step-by-step instructions on writing shell scripts and information about more sophisticated Bourne shell and C shell programming techniques, see Chapter 7, "The Bourne Shell Tutorial", and Chapter 8, "An Introduction to the C Shell".

# Communicating Electronically

As a UNIX system user, you can send messages or transmit information stored in files to other users who work on your system or another UNIX system. To do so, you must be logged in on a UNIX system that is capable of communicating with the UNIX 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** This command allows you to send messages or files to other UNIX 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 his or her convenience.

**uucp** This command is used to send files from one UNIX system to another. (Its name is an acronym for UNIX to UNIX system copy.) 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** These commands are used to send and retrieve files. You can use the **uuto** command to send a file to a public directory; when it is available, the recipient is notified by mail that the file has arrived. The recipient then can use the **uupick** command to copy the file from the public directory to a directory of choice.

**uux** This command 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 9 offers tutorials on each of these commands.

# Programming in the System

The UNIX system provides a powerful and convenient environment for programming and software development, using the C programming language and FORTRAN. The UNIX system provides some sophisticated tools designed to make software development easier and to provide a systematic approach to programming.

For information on the general topic of programming in the UNIX system environment, see the *IRIS-4D Programmer's Guide*. Besides supplementing texts on programming languages, the *IRIS-4D Programmer's Guide* provides tutorials on the following tools:

| | |
|---|---|
| **make** | maintains programs |
| **lex** | generates programs for simple lexical tasks |
| **yacc** | generates parser programs |

# The vi Editor

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. Also included in this chapter are some **ex**, or line editor commands. **ex** commands include the powerful global commands which allow you to change multiple occurrences of the same character string by issuing one command.

In order to use **vi** effectively you must know how to manipulate the text on your screen. To move through the file, **vi** allows you to scroll the text forward or backward, revealing the lines below or above the current window, as shown in Figure 5-1.

NOTE

Not all terminals have text scrolling capability; whether or not you can take advantage of **vi**'s scrolling feature depends on what type of terminal you have.

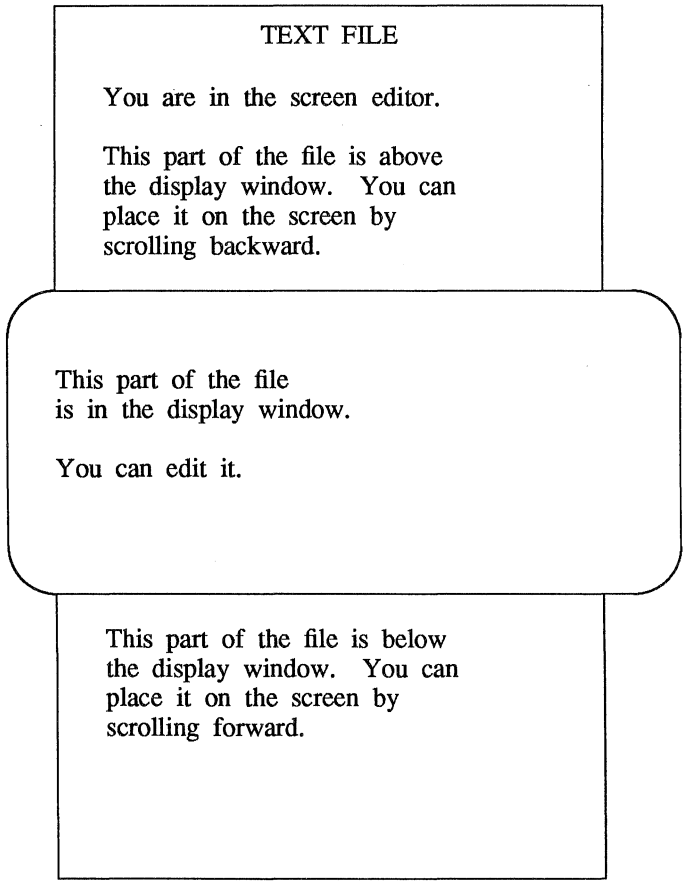



Figure 5-1: Displaying a File with a **vi** Window

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 **ex** 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

## Suggestions for Reading this Tutorial

As you read this tutorial, keep in mind the notation conventions described in the Preface. In the screens in this chapter arrows are also used to show the position of the cursor.

The commands discussed in each section are reviewed at the end of the section. At the end of some sections, exercises are given so you can experiment. The answers to all the exercises are at the end of this chapter.

A list of **vi** commands is found in a perforated quick-reference card at the beginning of this chapter. Tear it out and use it next to your workstation for quick access to the **vi** commands.

The best way to learn **vi** is by doing the examples and exercises as you read the tutorial. Log in on the UNIX system when you are ready to read this chapter.

# Getting Started

The UNIX system is flexible; it can run on many types of computers and 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 a given situation.

The UNIX system offers various optional features for using your terminal that you may 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 wrapmargin, or automatic carriage return, which is optional.

## Setting the Terminal Configuration

Before you enter **vi**, you must set your terminal configuration. This simply means that you tell the UNIX system what type of terminal you are using. This is necessary because the software for **vi** is executed differently on different terminals.

Each type of terminal has several code names that are recognized by the UNIX system. Appendix D, "Setting Up the Terminal," 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 your UNIX system supports. It is a good idea to check with your local system administrator for the most up-to-date list of available terminal types.

To set your terminal configuration, type

**TERM=***terminal_name***<return>**
**export TERM<return>**
**tput init<return>**

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 UNIX system programs whose execution depends on the type of terminal being used.

The **tput** command on the third line initializes (sets up) the software in your terminal so that it functions properly with the UNIX system. It is essential to run the **tput init** command when you are setting your terminal configuration because terminal functions such as tab settings will not work properly unless you do.

Do not experiment by entering names for terminal types other than your terminal. This might confuse the UNIX 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** in the Bourne shell. (For details, see Chapter 7.) **.login** controls the environment in the C shell. (For details, see Chapter 8.)

If you specify the setting for your terminal configuration in your **.profile**, your terminal will be configured automatically every time you log in. You can do this by adding the **TERM** assignment, **export** command, and **tput** command to your **.profile**. (For detailed instructions, see Chapter 7.)

## Setting the Automatic Return


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 **<return>** 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 with 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*<return>**

*n* represents the number of characters from the righthand side of the screen where you want an automatic carriage return to occur. For example, say you want a carriage return at 20 characters from the righthand side of the screen. Type

**wm=20<return>**

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<return>**

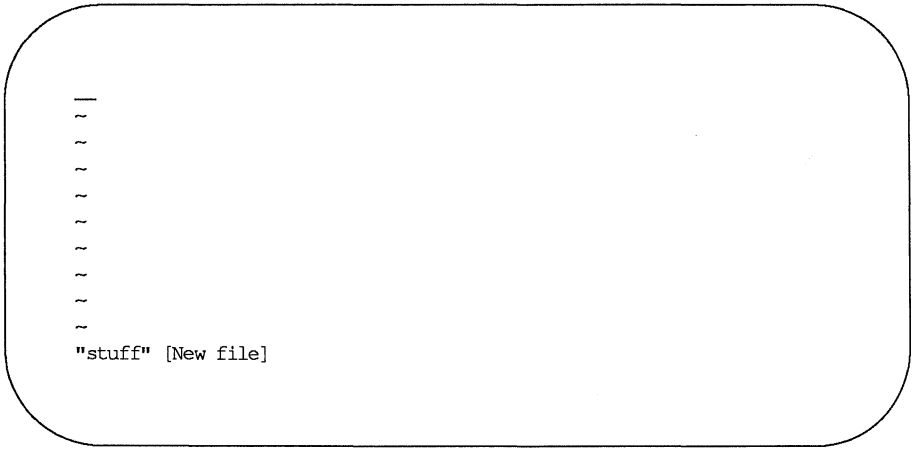
**vi** will report the terminal type and the wrapmargin, as well as any other options you may have specified. You can also use the **:set** command to create or change the wrapmargin option. Try experimenting with it.

# Creating a File

To enter **vi** type **vi** and the name of the file you want to create or edit.

**vi** *filename*<**return**>

For example, say 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 shows 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.) Every other line is marked with a ~ (tilde), the symbol for an empty 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.

```
% vi stuff<return>
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<return>**

Then set the correct terminal configuration.

## How to Create Text: the Append Mode

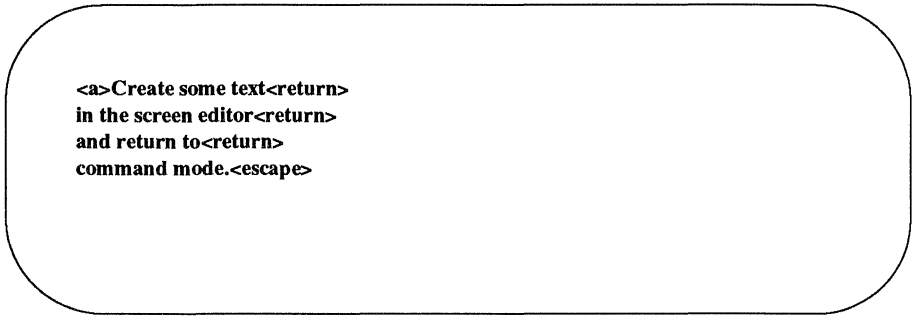
If you have successfully entered **vi**, you are in command mode and **vi** is waiting for your commands. How do you create text?

- Press the A key (**<a>**) to enter the append mode of **vi**. (Do not press **<return>**.) You can now add text to the file. (An A is not printed on the screen.)
- Type in some text.
- To begin a new line, press **<return>**.

  If you have specified the wrapmargin option in a **.exrc** file, you will get a new line whenever you get an automatic return (see "Setting the Automatic Return").

## How to Leave Append Mode

When you finish creating text, press **<escape>** to leave append mode and return to command mode. Then you can edit any text you have created or write the text in the buffer to a file.




If you press **<escape>** and a bell sounds, you are already in command mode. The text in the file is not affected by this, even if you press **<escape>** several times.

# Editing Text: The 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 in a line. These commands, along with commands for deleting and adding text, are introduced in this section.

## How to Move the Cursor

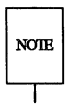
To edit your text, you need to move the cursor to the point on the screen where you will begin the correction. This is easily done with four keys that are grouped together on the keyboard: h, j, k, and l.

- **<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.

**: q<return>**




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 **<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. Type

**<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 will sound and the cursor will remain on the current line.

Now move the cursor down 35 lines. Type

**<35j>**

**vi** will clear and redraw the screen. The cursor will be on the thirty-fifth 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 will sound and the cursor will remain 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 or insert mode and are adding text to your file. If the motion command letters appear on the screen, you are still in append mode. Press **<escape>** to return to command mode and try the commands again.

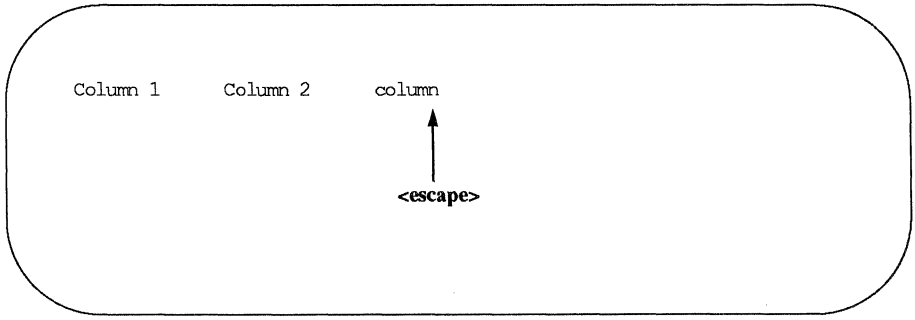
## Moving the Cursor to the Right or Left

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.

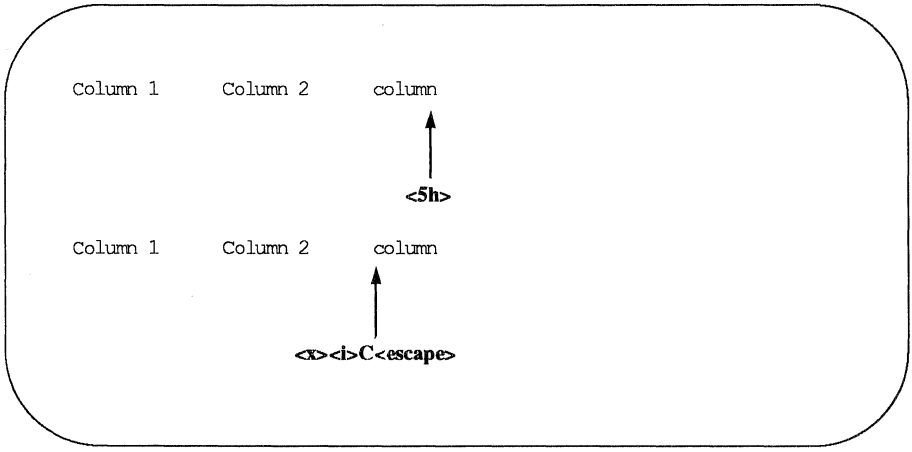
| | |
|---|---|
| **<space bar>** | move the cursor one character to the right |
| **<*n*space bar>** | move the cursor *n* characters to the right |
| **<backspace>** | move the cursor one character to the left |
| **<*n*backspace>** | move the cursor *n* characters to the left |

Try typing a number before the command key. Notice that the cursor moves the specified number of characters to the left or right. In the example below, the cursor movement is shown by the arrows.

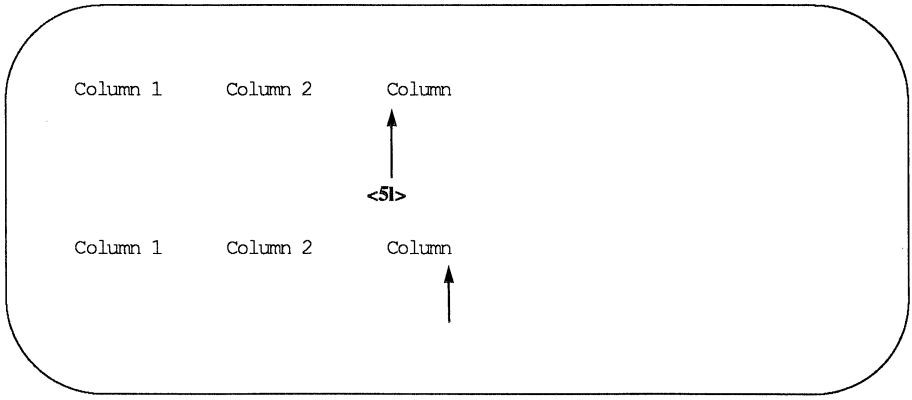
To move the cursor quickly to the right or left, prefix a number to the command. For example, suppose you want to create four columns in your screen. After you've finished typing the headings for the first three columns, you notice a typing mistake.




You want to correct your mistake before continuing. Exit insert mode and return to command mode by pressing **<escape>**; the cursor will move to the **n**. Then use the **<h>** command to move back five spaces.



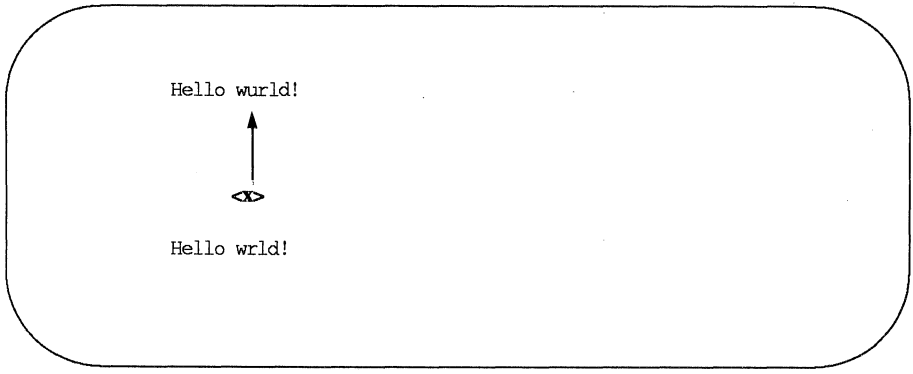
Erase the c by typing **<x>**. Then change to insert mode (**<i>**), enter a **C**, followed by pressing **<escape>**. Use the **<l>** motion command to return to your earlier position.



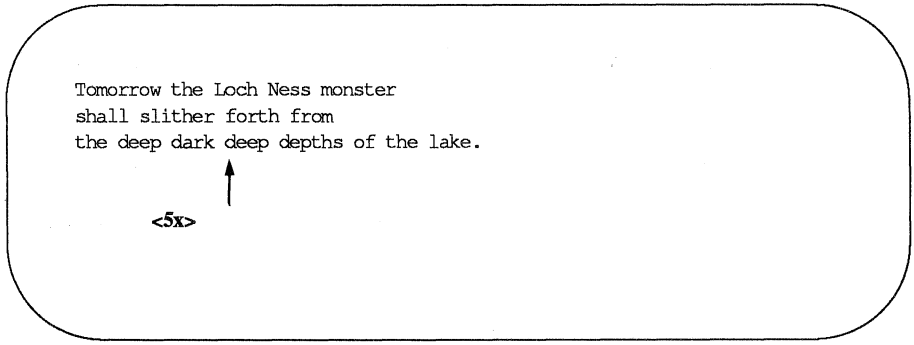

## How to Delete Text

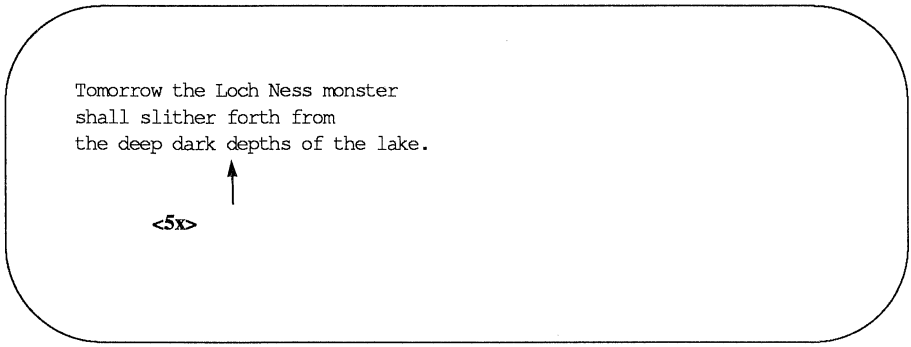
If you want to delete a character, move the cursor to that character and press the **<x>**. Watch the screen as you do so; the character will disappear and the line will readjust to the change. To erase three characters in a row, press **<x>** three times. In the following examples, the arrows under the letters show the cursor position.

| | |
|---|---|
| **<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).

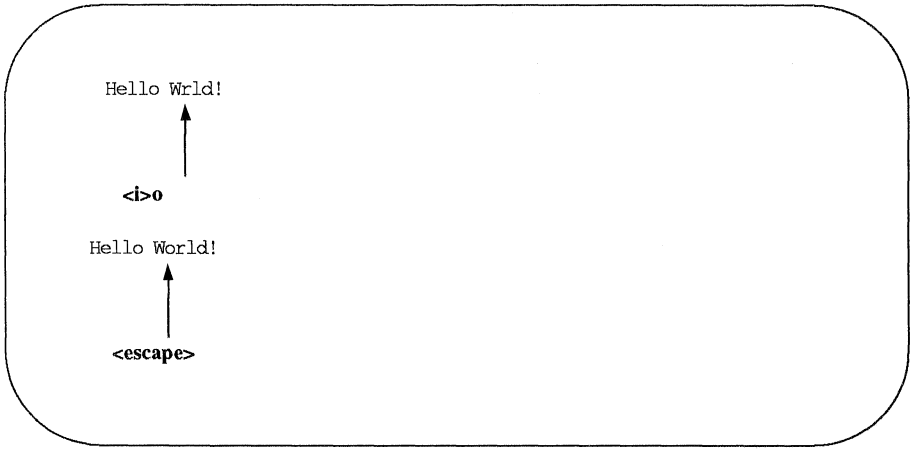





Notice that **vi** adjusts the text so that no gap appears in place of the deleted string. If, as in this case, the string you want to delete happens to be a word, you can also use the **vi** command for deleting a word. This command is described later in the section "Word Positioning."

## How to Add 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 will appear 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 will move right to make room for your new text. The **vi** editor will continue to accept the characters you type until you press **<escape>**. If necessary, the original characters will even wrap around onto the next line.




You can use the append command in the same way. The only difference is that the new text will appear 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.<return>
I am adding text to<return>
a temporary buffer and<return>
now it is perfect.<return>
I want to write this file,<return>
and return to the shell.<escape><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**.

```
<a>This is a test file.<return>
I am adding text to<return>
a temporary buffer and<return>
now it is perfect.<return>
I want to write this file,<return>
and return to the shell.<escape>
~
~
~
~
~
:wq<return>
%
```

Figure 5-2 summarizes the basic commands you need to enter and use **vi**.

| Command | Function |
|---|---|
| **TERM=***terminal_name*<br>**export TERM** | set the terminal configuration |
| **tput init** | 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 |
| **<return>** | carriage return |
| **<escape>** | 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** |

Figure 5-2: Summary of Commands for the **vi** Editor

# 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. Reenter **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 section, "Positioning the Cursor in Undisplayed 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 **<escape>** to make sure that you are in command mode rather than 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

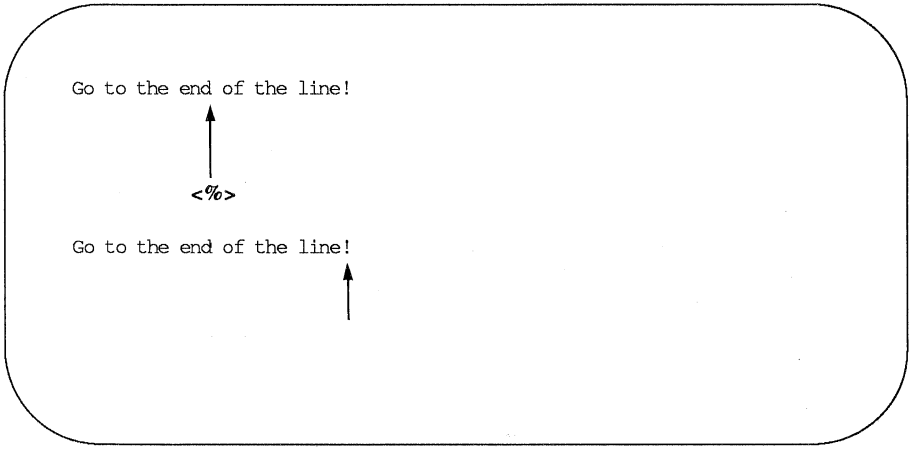
The first method was discussed earlier in this chapter under "Moving the Cursor to the Right or Left." The following sections describe the other two methods.

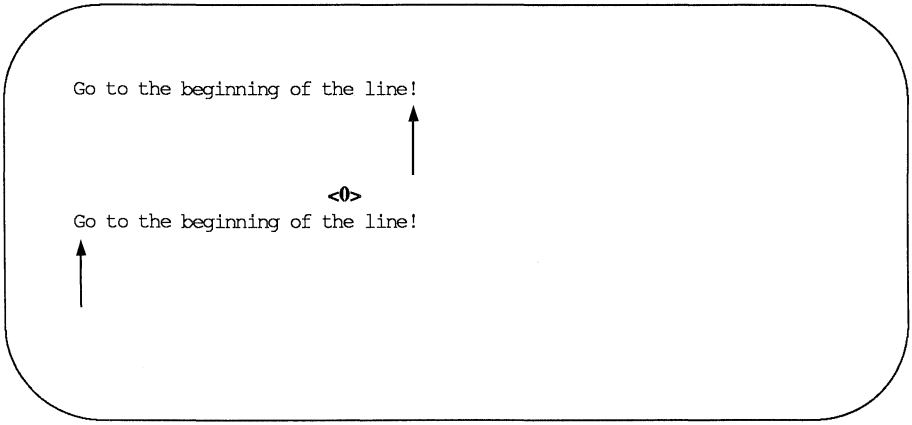
## Moving the Cursor on a Line

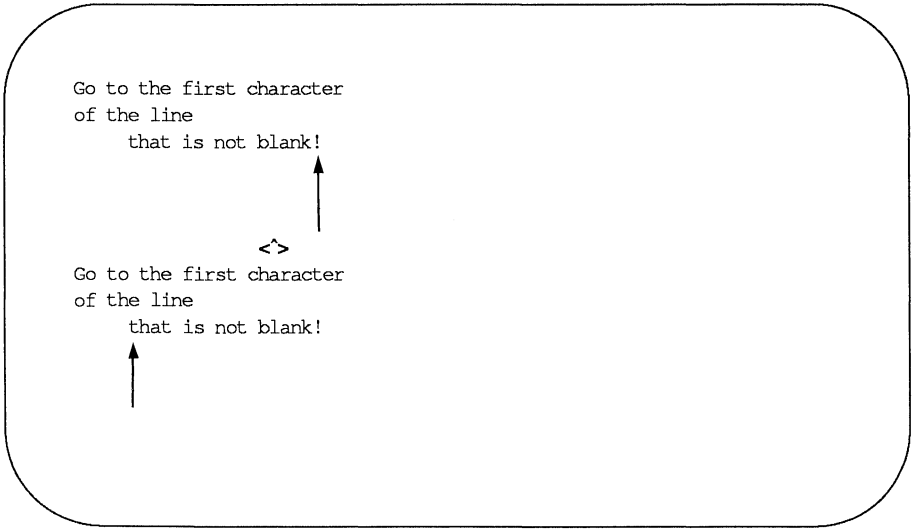
The second method of positioning the cursor on a line is by using one of three commands that put the cursor on the first or last character of a line.

| | |
|---|---|
| <%> | 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 beginning of the line!
<0>
Go to the beginning of the line!


Go to the first character
of the line
that is not blank!
<^>
Go to the first character
of the line
that is not blank!

## 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*.

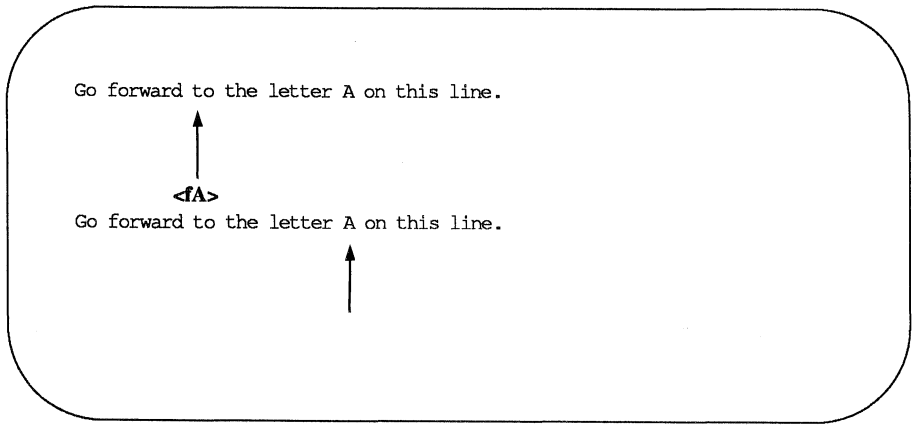
**<t*x*>** Move the cursor right to the character just before the specified character *x*.

**<T*x*>** 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.




Try the search commands on one of your files.

# Line Positioning

Besides the <j> and <k> commands that you have already used, the <+>, <->, and <return> 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 non-blank 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 will move up 13 lines. If some of those lines are above the current window, the window will scroll up to reveal them.

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 will sound, telling you that you have made a mistake, and the cursor will remain on the current line.

## The Plus Sign Motion Command

The plus sign command (<+>) or the <return> 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+>

If some of those lines are below the current screen, the window will scroll down to reveal them.

Now try to do the same thing by pressing <return>. Were the results the same as when you pressed the + key?

# Word Positioning

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

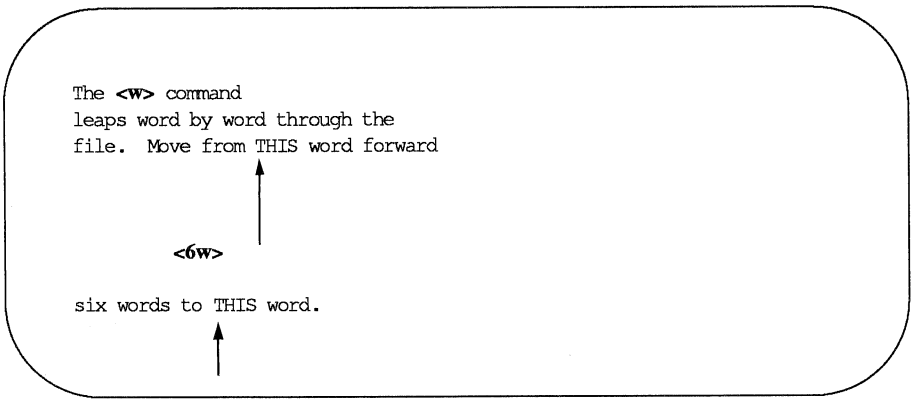
the beginning or end of a word. Punctuation before or after a blank is considered a word. The beginning or end of a line is also a delimiter.

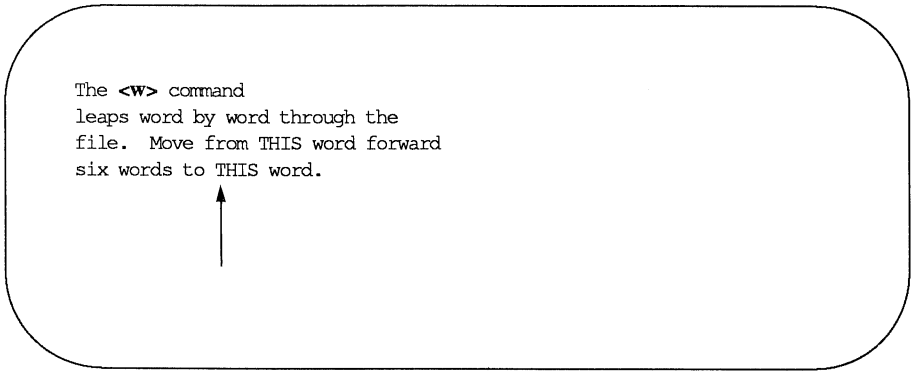
The uppercase commands (<W>, <B>, and <E>) treat punctuation as part of the word; words are delimited by blanks and new lines only.

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>.

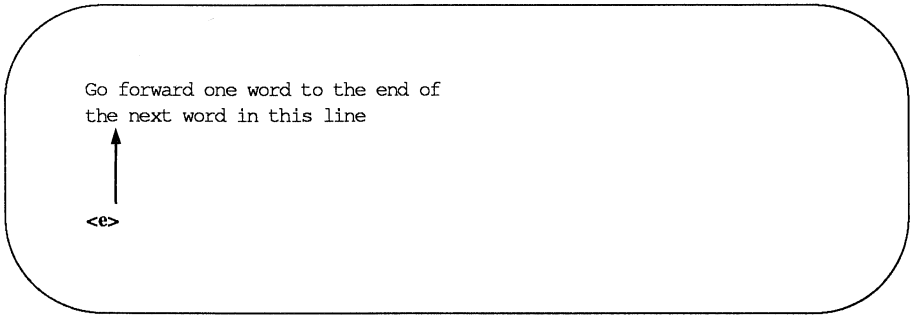
**<*n*w>** Move the cursor forward *n* number of words to the first character of that word. The end of the line does not stop the movement of the cursor; instead, the cursor wraps around and continues counting words from the beginning of the next line.

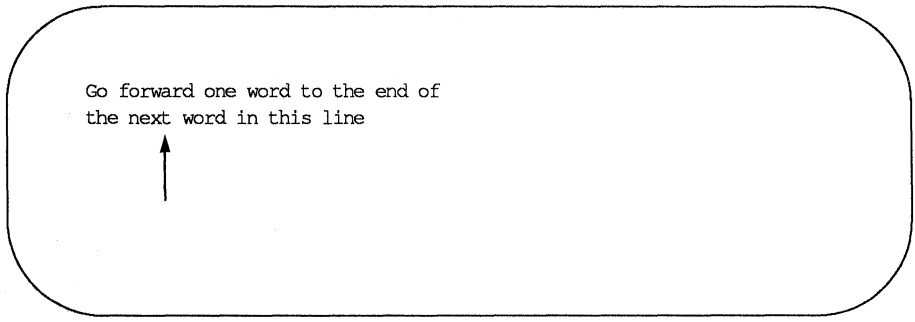


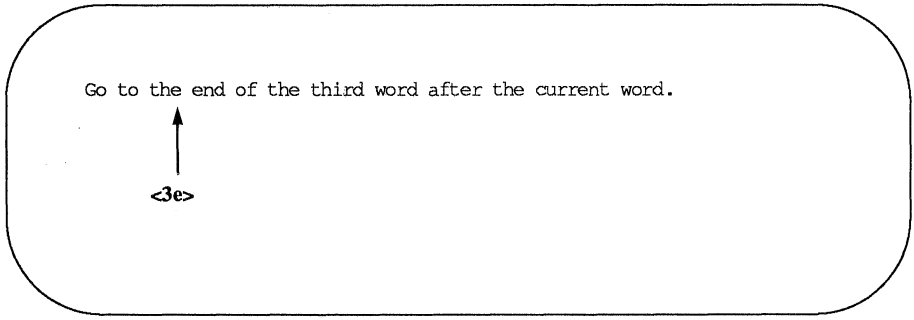


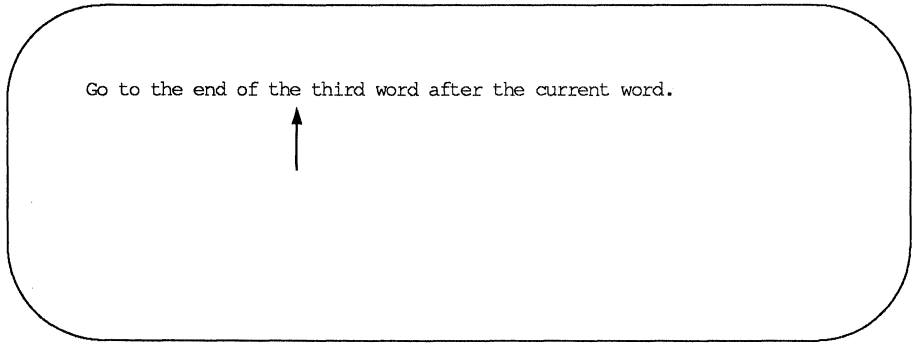

**<W>** Ignore all punctuation and move the cursor forward to the word after the next blank.

**<e>** Moves the cursor forward in the line to the last character in the next word.





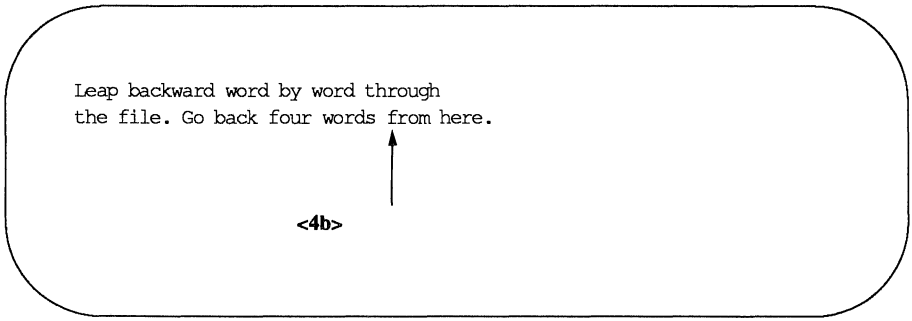



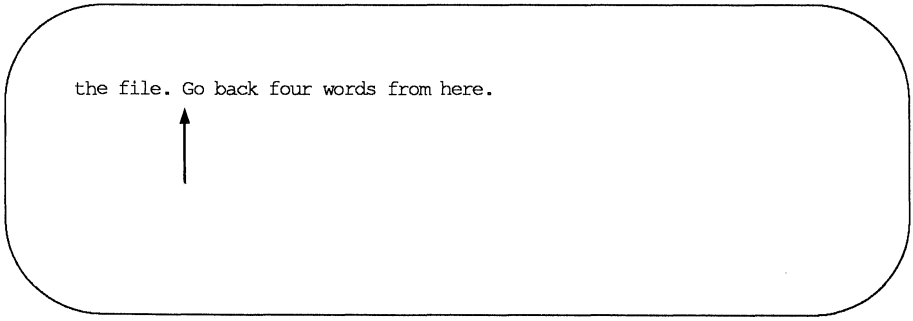

**<E>** Ignores all punctuation except blanks, delimiting words only by blanks.

**<b>** Move the cursor backward in the line to the first character of the previous word.

**<*n*b>** Move 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 new lines. It treats all other punctuation as letters of a word.



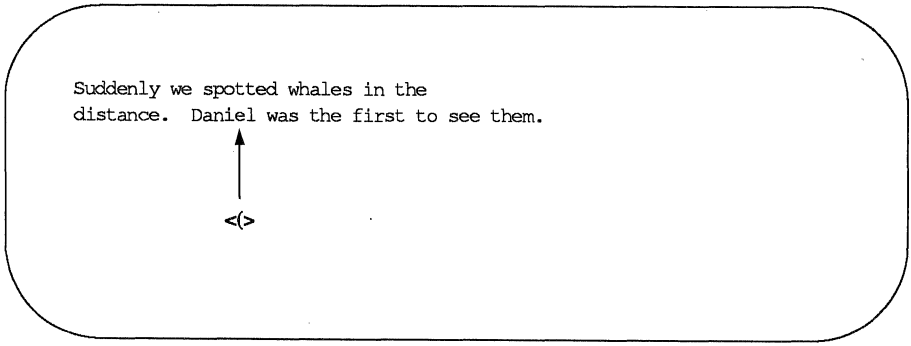
## Positioning the Cursor by Sentences

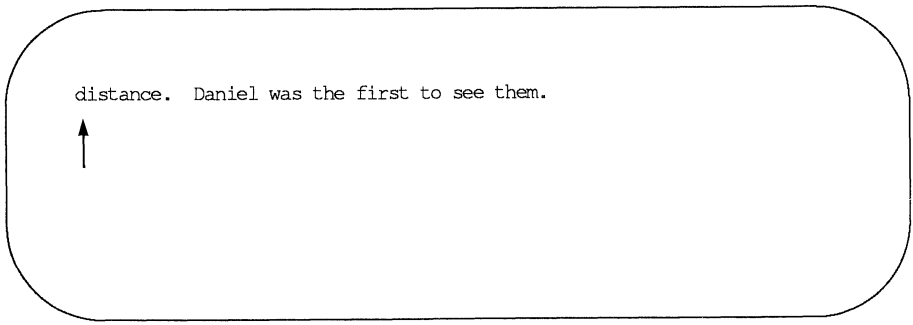
The **vi** editor also recognizes sentences. In **vi** a sentence ends in ! or . or ?. If these delimiters appear in the middle of a line, they must be followed by two blanks for **vi** to recognize them. You should get used to the **vi** convention of recognizing two blanks after a period as the end 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.

- **< ( >** Move the cursor to the beginning of the current sentence.
- ***< n( >*** Move the cursor to the beginning of the *n*th sentence above the current sentence.
- **< ) >** Move the cursor to the beginning of the next sentence.
- ***< n) >*** Move the cursor to the beginning of the *n*th sentence below the current sentence.

The example in the following screens 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)>**

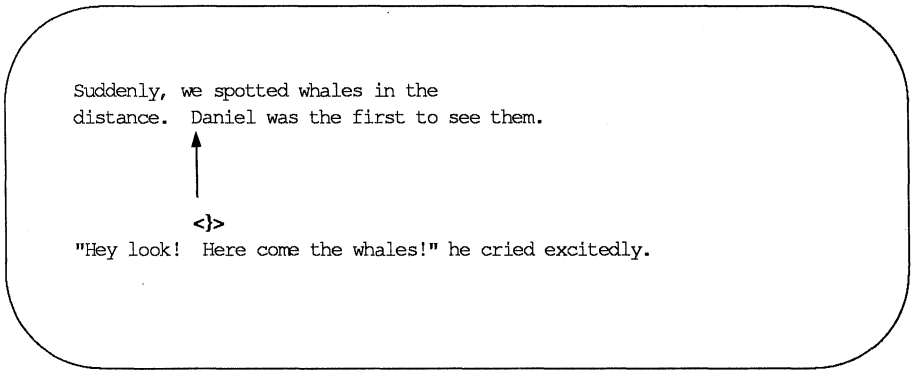
Did the cursor move the correct number of sentences?

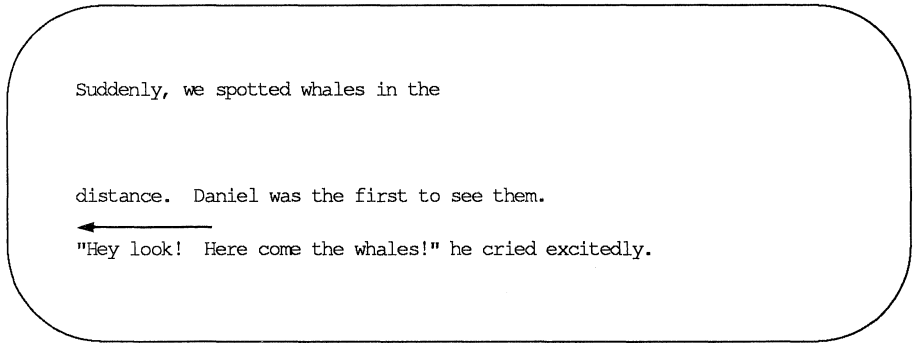
## Positioning the Cursor by Paragraphs

**vi** recognizes paragraphs 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.

| | |
|---|---|
| **<{>** | Move the cursor to the beginning of the current paragraph, which is delimited by a blank line above it. |
| **<n{>** | Move the cursor to the beginning of the *n*th paragraph above the current paragraph. |
| **<}>** | Move the cursor to the beginning of the next paragraph. |
| **<n}>** | Move the cursor to the *n*th paragraph below the current line. |

The following two screens show how the cursor can be moved to the beginning of another paragraph.

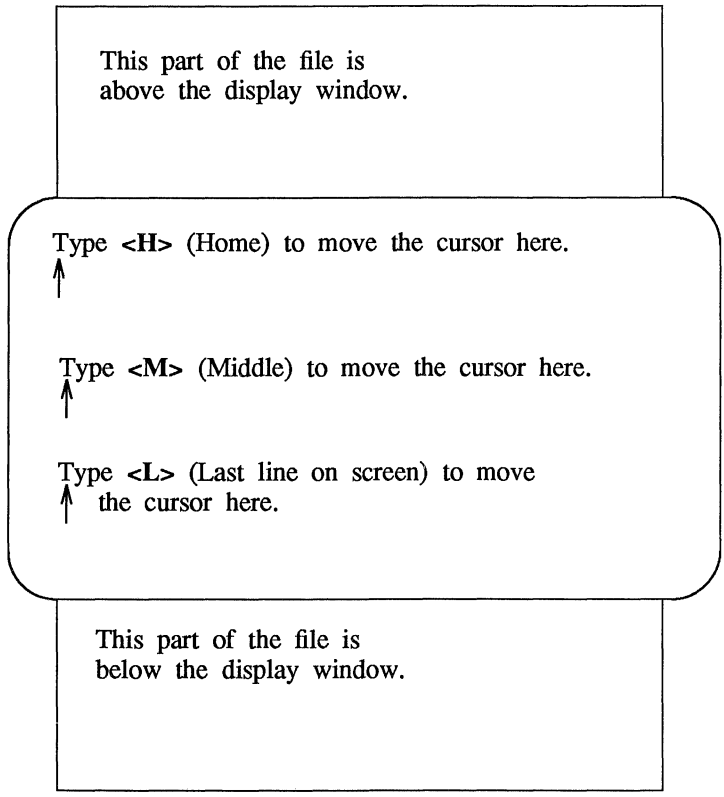



## Positioning in the Window

The **vi** editor also provides three commands that help you position the cursor in the window. Try out each command. Be sure to type them in uppercase.

| | |
|---|---|
| **<H>** | Move the cursor to the first line on the screen. |
| **<M>** | Move the cursor to the middle line on the screen. |
| **<L>** | Move the cursor to the last line on the screen. |




Figures 5-3 through 5-6 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 Figure 5-7, later in the chapter.)

| 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. |

Figure 5-3: Summary of **vi** Motion Commands (Sheet 1 of 4)

| 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. |
| **<return>** | Move the cursor down to the beginning of the next line. |

Figure 5-4: Summary of **vi** Motion Commands (Sheet 2 of 4)

| 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. |

Figure 5-5: Summary of **vi** Motion Commands (Sheet 3 of 4)

| | |
|---|---|
| **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. |

Figure 5-6: Summary of **vi** Motion Commands (Sheet 4 of 4)

# Positioning the Cursor in Undisplayed 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 those 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

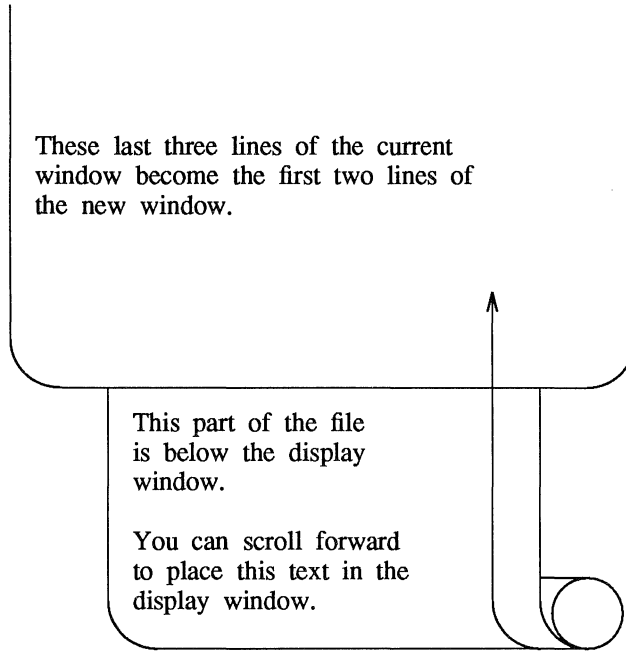
## Scrolling the Text

Four commands allow you to scroll the text of a file. The **<ctrl-f>** and **<ctrl-d>** commands scroll the screen forward. The **<ctrl-b>** and **<ctrl-u>** commands scroll the screen backward.

### The <ctrl-f> Command

The **<ctrl-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:

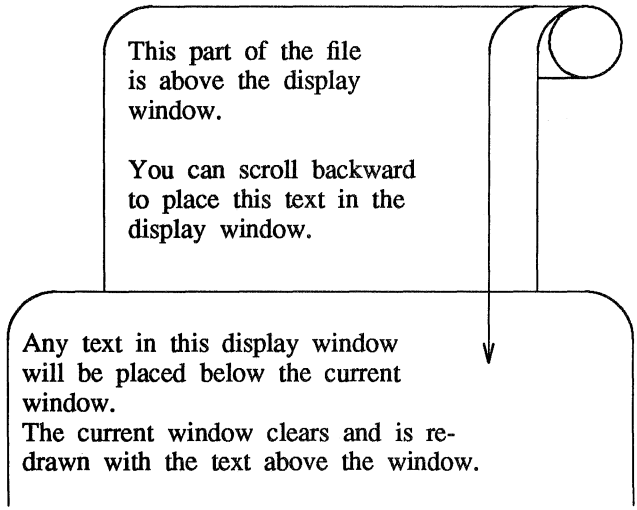



## The <ctrl-d> Command

The **<ctrl-d>** command scrolls down a half screen to reveal text below the window. When you type **<ctrl-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 will sound.

## The <ctrl-b> Command

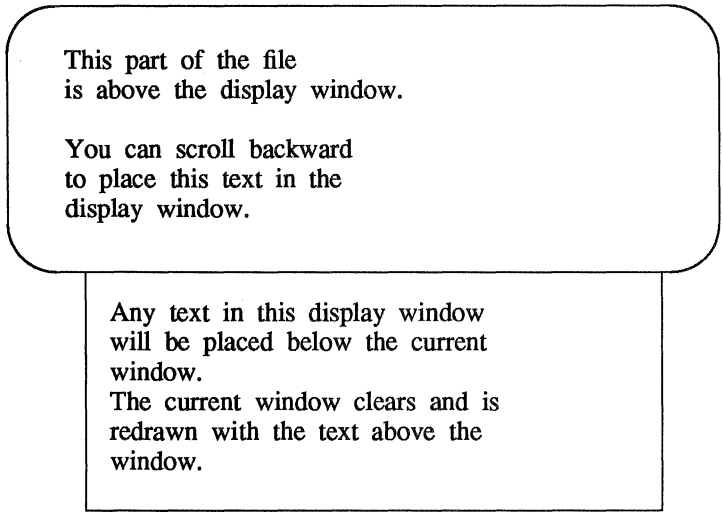
The **<ctrl-b>** command scrolls the screen back a full window to reveal the text above the current window. To do this, **vi** clears the screen and redraws the window with the text that is above the current screen. Unlike the **<ctrl-f>** command, **<ctrl-b>** does not leave any reference lines from the previous window. If there are not enough lines above the current window to fill a full new window, a bell will sound and the current window will remain on the screen.




Now try scrolling backward. Type

**<ctrl-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 <ctrl-u> Command

The **<ctrl-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:

**<ctrl-u>**

When the cursor reaches the top of the file, a bell sounds to notify you that the file cannot scroll further.

## Go to a Specified 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

## Line Numbers

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 **<ctrl-g>**. The **<ctrl-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.

        <ctrl-g>

There are several more lines in the
buffer.

"file.name" [modified] line 36 of 116 --34%--
```

## Searching for a Pattern of Characters

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 command line

**/Hello world<return>**

finds the next occurrence in the buffer of the words **Hello world** and puts the cursor under the **H**.

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 command line

**?data set design<return>**

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**.

These search commands do not wrap around the end of a line while searching for two words. For example, say 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 will 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 will find the pattern.

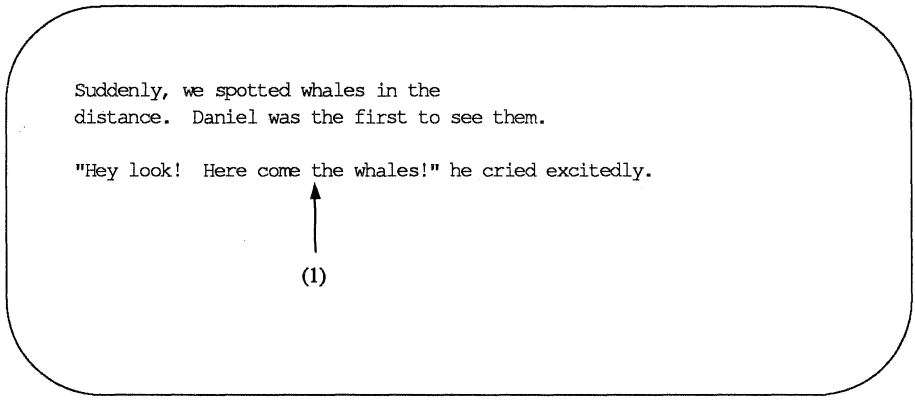
The **<n>** and **<N>** commands allow you to continue searches you have requested with */pattern* or **?***pattern* without retyping them.

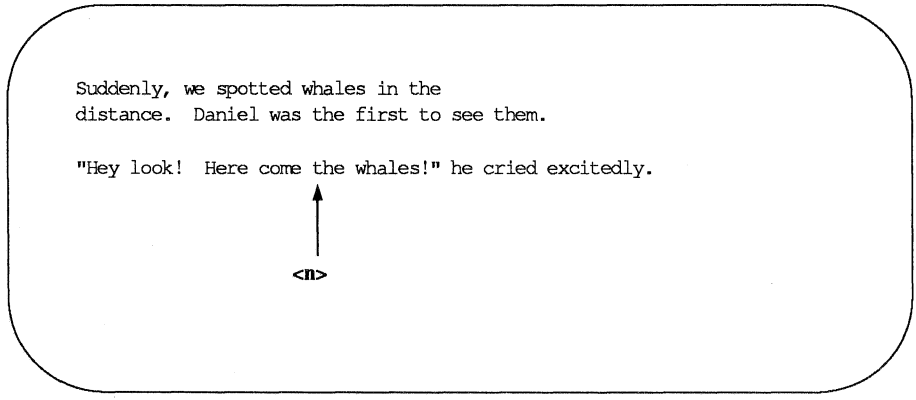
**<n>** Repeat the last search command.

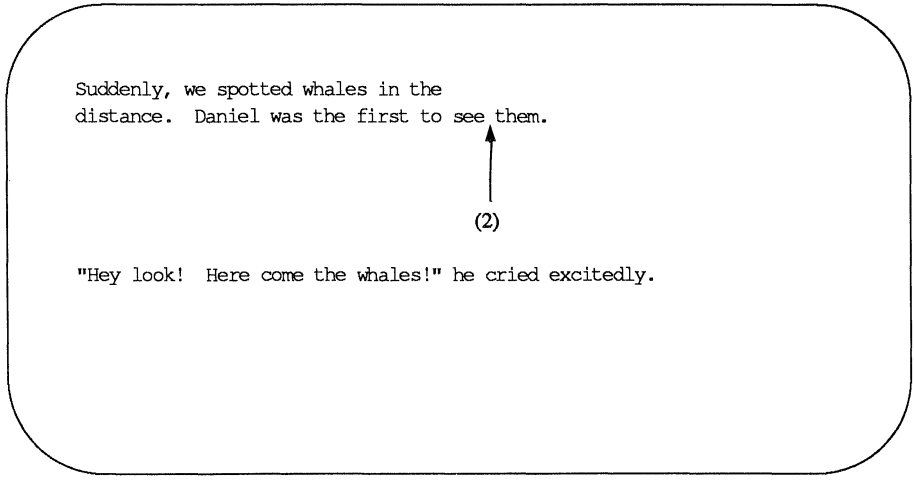
**<N>** Repeat the last search command in the opposite direction.

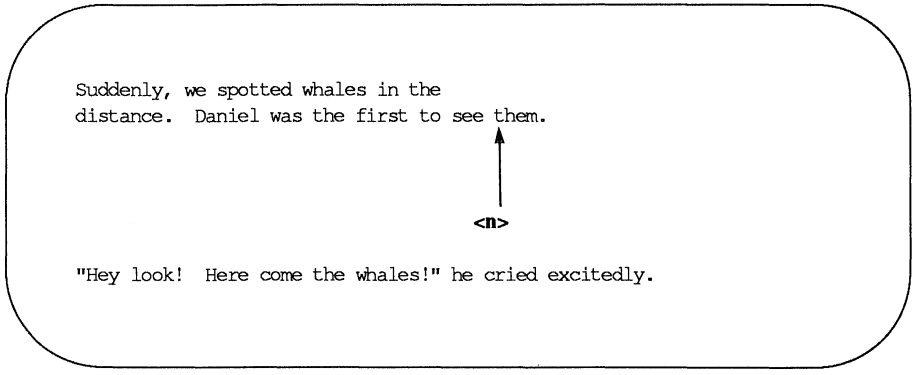
For example, say you want to search backward in the file for the three-letter pattern `the`. Initiate the search with **?***the* and continue it with **<n>**. The following screens offer a step-by-step illustration of how the **<n>** searches backward through the file and finds four occurrences of the character string `the`.

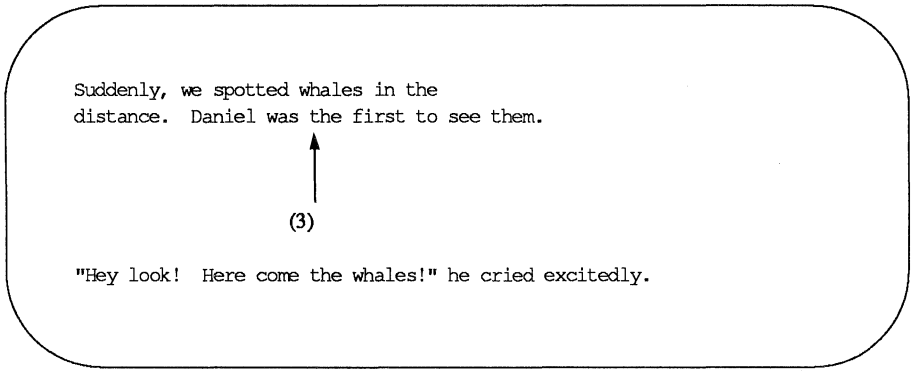
```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.

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



?the
```

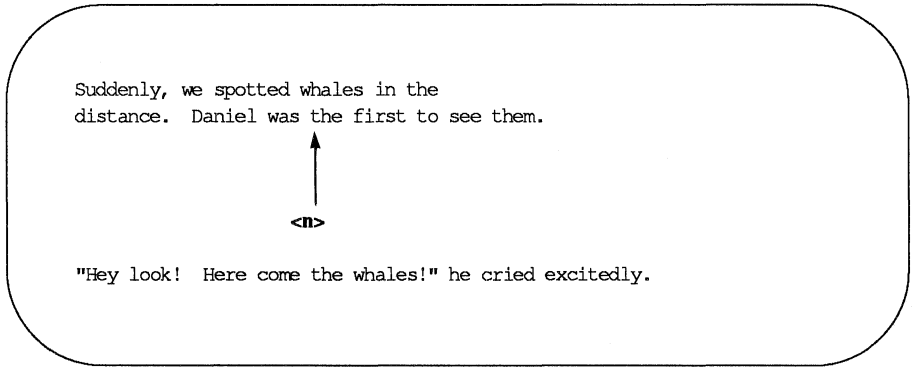

Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
"Hey look! Here come the whales!" he cried excitedly.
(1)

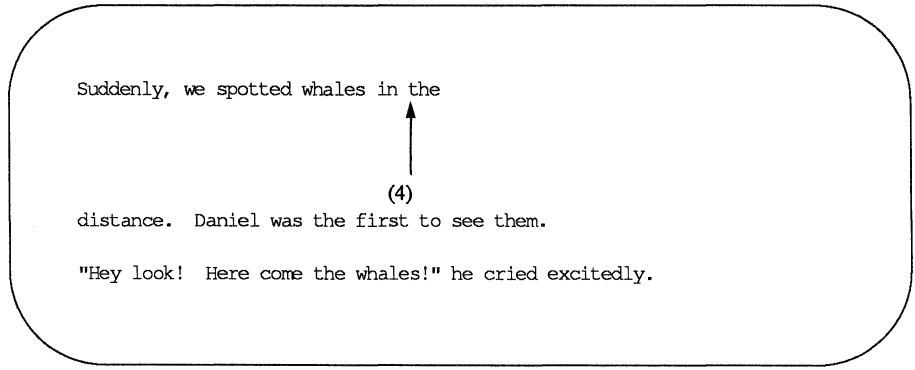

Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
"Hey look! Here come the whales!" he cried excitedly.
<n>


Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
(2)
"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.
(3)
"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
(4)
distance. Daniel was the first to see them.
"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*.

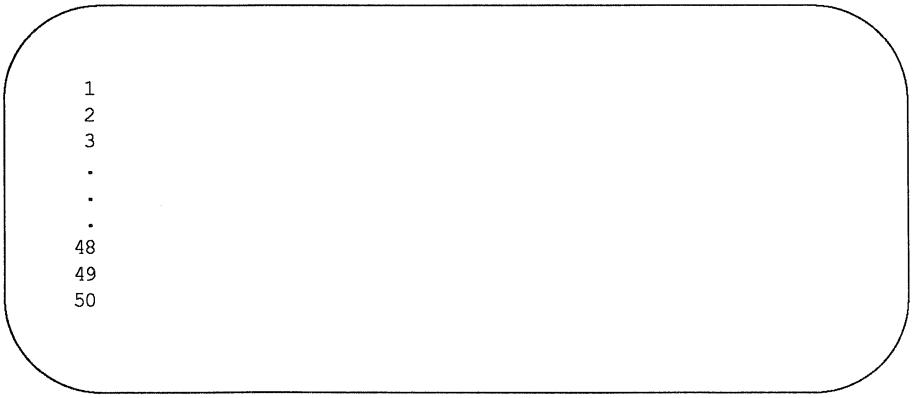
Figure 5-7 summarizes the **vi** commands for moving the cursor by scrolling the text, specifying a line number, and searching for a *pattern*.

| Scrolling | |
|---|---|
| **<ctrl-f>** | Scroll the screen forward a full window, revealing the window of text below the current window. |
| **<ctrl-d>** | Scroll the screen down a half window, revealing lines below the current window. |
| **<ctrl-b>** | Scroll the screen back a full window, revealing the window of text above the current window. |
| **<ctrl-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. |
| **<ctrl-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. |

Figure 5-7: Summary of Additional **vi** Motion Commands

# 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.




2-2. Try using each of the scroll commands, noticing how many lines scroll through the window. Try the following:

**<ctrl-f>**
**<ctrl-b>**
**<ctrl-u>**
**<ctrl-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. 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 number 8. Find the next occurrence of 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 **<escape>**.

## 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 **<escape>**. Then compare the **<a>** command to the **<A>** command.

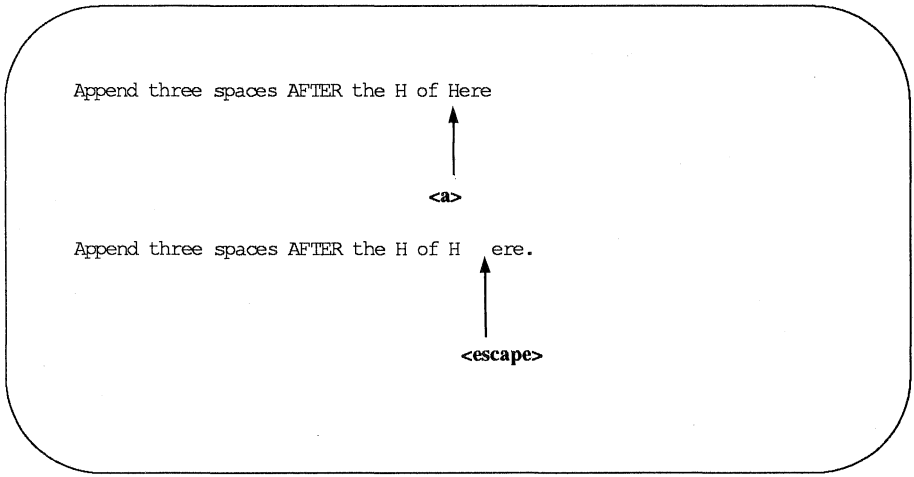
## Inserting Text

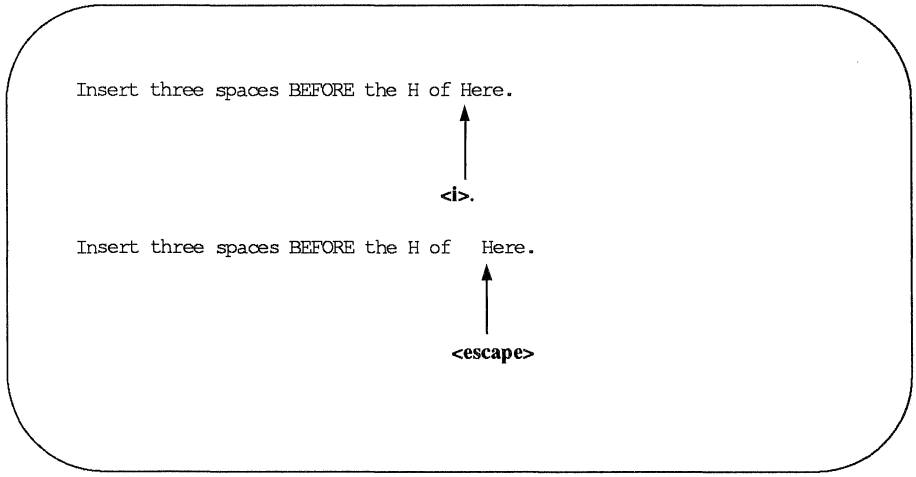
**<i>** insert text before the cursor

**<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 **<escape>**.

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.



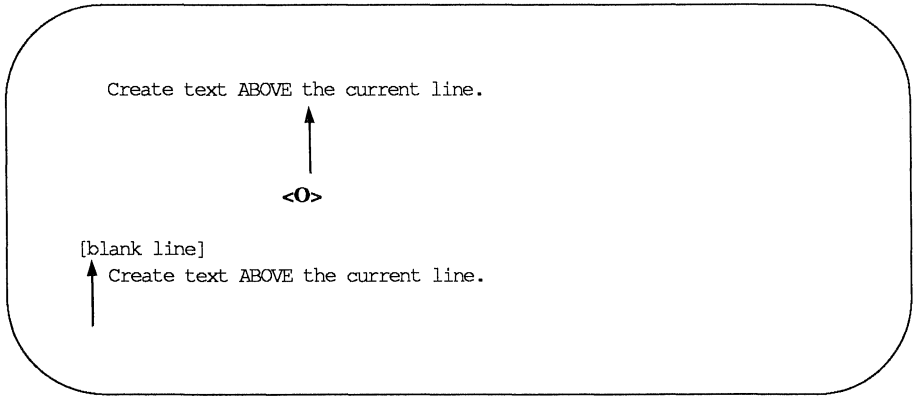

Notice that in both cases, the user has left text input mode by pressing **<escape>**.

# Opening a Line for Text

**<o>** Create 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>** Create 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 blank line 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.



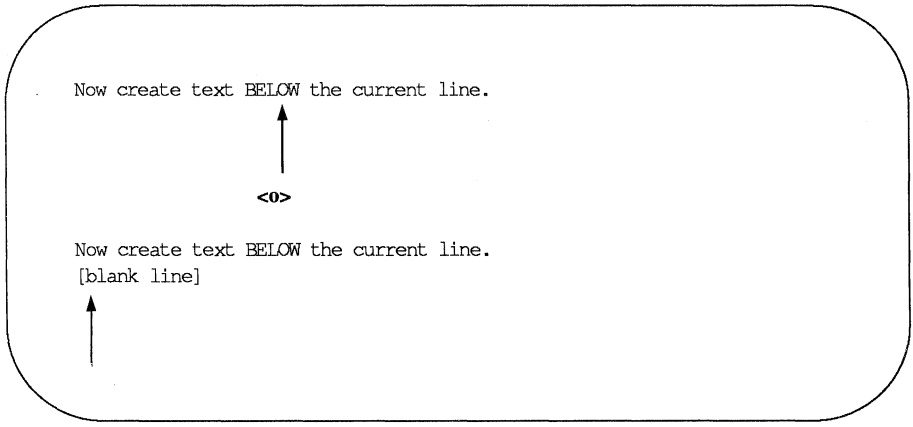



Figure 5-8 summarizes the commands for creating and adding text with the **vi** editor.

| 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. |
| **<escape>** | Return **vi** to command mode from any of the above text input modes. |

Figure 5-8: Summary of **vi** Commands for Creating Text

# 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

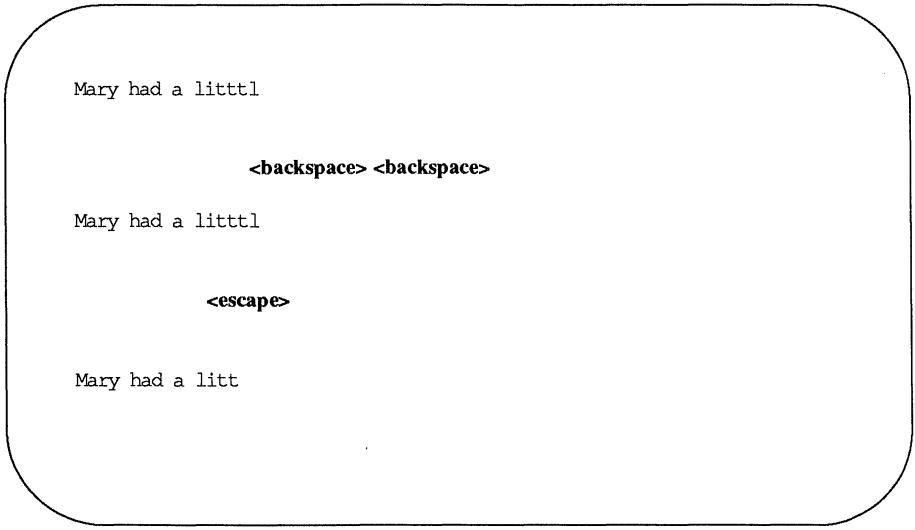
You can delete text with various commands in command mode, and undo the entry of small amounts of text in text input mode. In addition, you can undo entirely the effects of your most recent command.

## 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.

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 **<escape>** to return to command mode.

In the following example, the arrows represent the cursor.




Notice that the characters are not erased from the screen until you press **<escape>**.

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.

When you type **<ctrl-w>**, the cursor backs up over the word last typed and waits on the first character. It does not literally erase the word until you press **<escape>** or enter new characters over the old ones. The **@** sign behaves in a similar manner except that it removes all text you have typed on the current line since you last entered input mode.

## Undo the Last Command

Before you experiment with the delete commands, you should try the **u** command. This command undoes the last command you issued.
**Undo the last command."** ljust

| 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. |
| **<escape>** | Return **vi** to command mode from any of the above text input modes. |

Figure 5-8: Summary of **vi** Commands for Creating Text

**<u>** undo the last command

**<U>** restore the current line to its state before you changed it

If you delete lines by mistake, type **<u>**; your lines will reappear on the screen. If you type the wrong command, type **<u>** and it will be nullified. The **<U>** command will nullify 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 will undo the first; your undo will be undone! For example, say you delete a line by mistake and restore it by typing **<u>**. Typing **<u>** a second time will delete the line again. Knowing this command can save you a lot of trouble.

# Delete Commands in Command Mode

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 some components of the command format show that those components are optional.

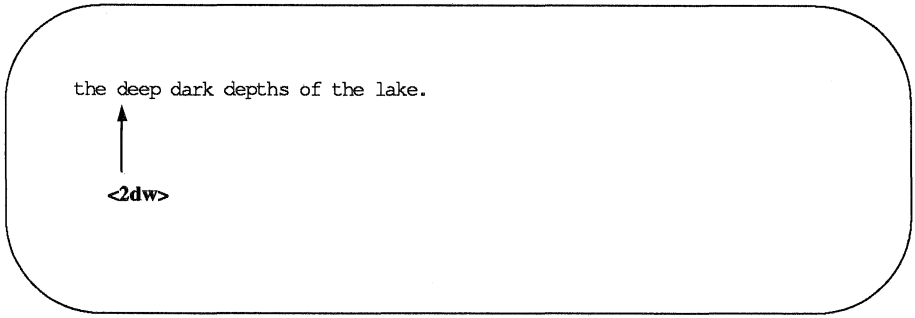
All delete commands issued in command mode immediately remove unwanted text from the screen and redraw the affected part of the screen.

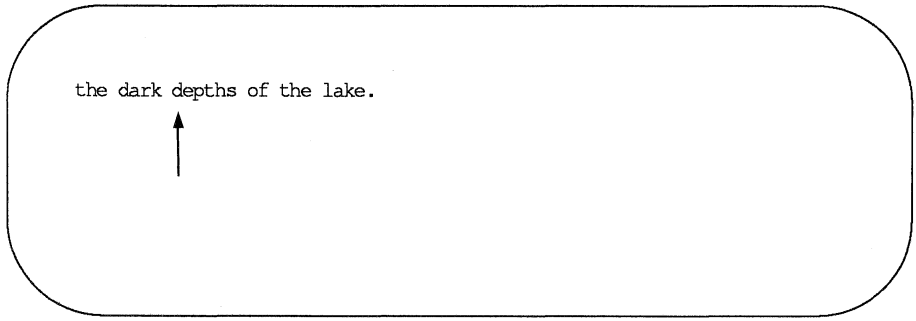
The delete command follows the general format of a **vi** command.

[*number*]**d***text_object*

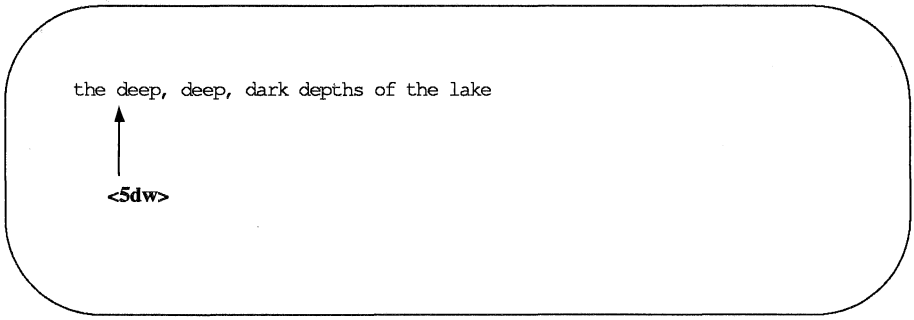
## Deleting Words

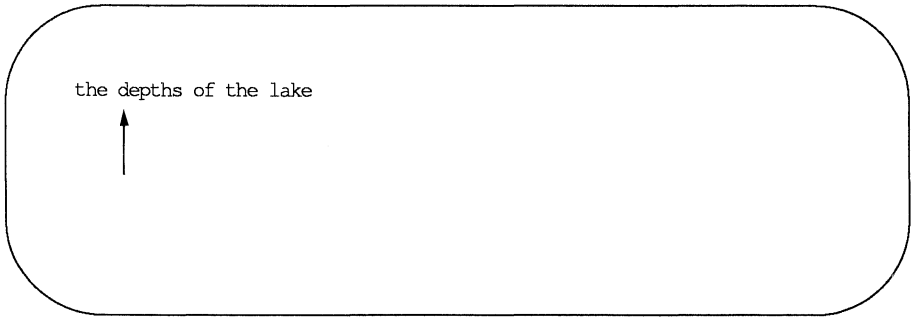
You can delete a word or part of a word with the **<dw>** command. Move the cursor to the first character to be deleted and type **<dw>**. The character under the cursor and all subsequent characters in that word will be erased.




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>**.






## 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

**<10dd>**

will erase 10 lines. If you delete more than a few lines, **vi** will display this 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 will sound and no lines will be 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.

Figure 5-9 summarizes the **vi** commands for deleting text.

| Command | Function |
|---|---|
| **For INSERT Mode:** | |
| **<backspace>** | Delete the current character. |
| **<ctrl-h>** | Delete the current character. |
| **<ctrl-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. |

Figure 5-9: Summary of Delete Commands

# 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>** Replace 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 **<escape>**.

**<*n*r>** Replace *n* characters with the same letter. This command automatically terminates after the *nth* character is replaced. It does not need to be followed by **<escape>.**

**<R>** Replace 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 **<escape>**.

**<s>** Delete the character shown by the cursor and append text. End the text input mode by pressing **<escape>.**

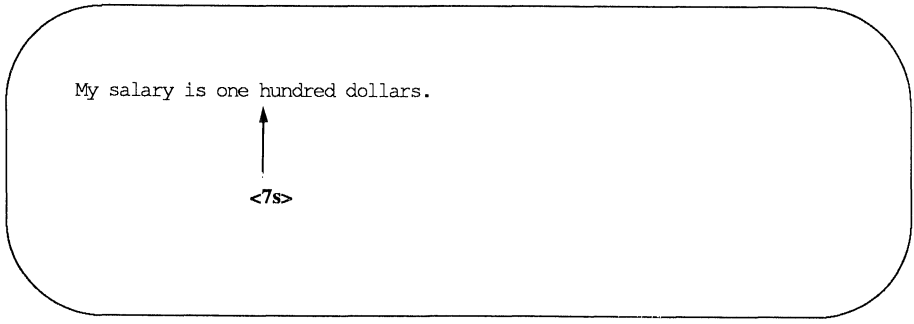
**<*n*s>** Delete *n* characters and append text. End the text input mode by pressing **<escape>.**

**<S>** Replace all the characters in the line.

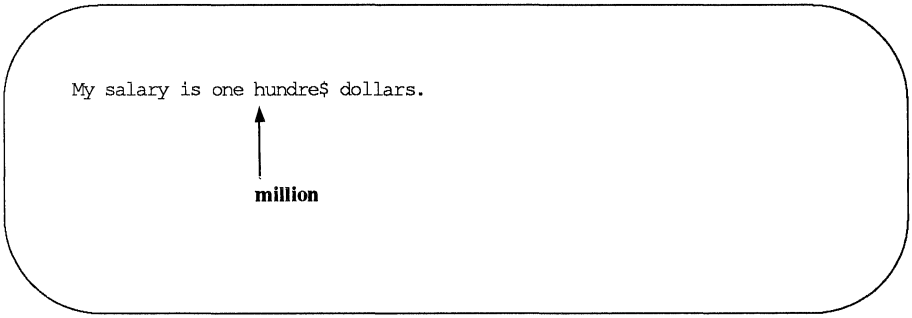
When you enter the **<s>** command, the last character in the string of characters to be replaced is overwritten by a `$` sign. The characters are not erased from the screen until you type over them, or leave text input mode.

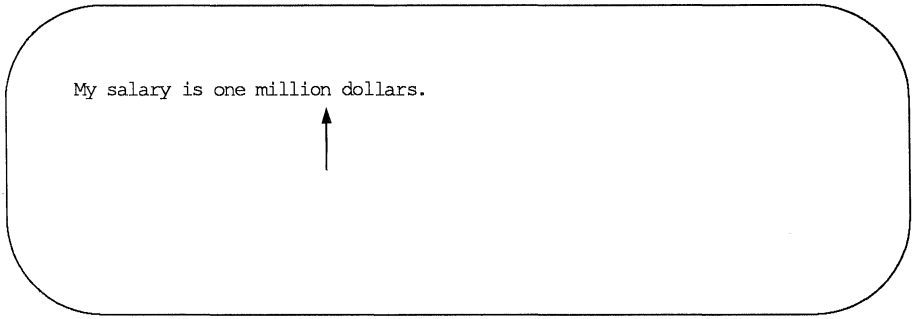
Notice that you cannot use an argument with either **<r>** or **<s>**. Did you try?

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 **<escape>**.

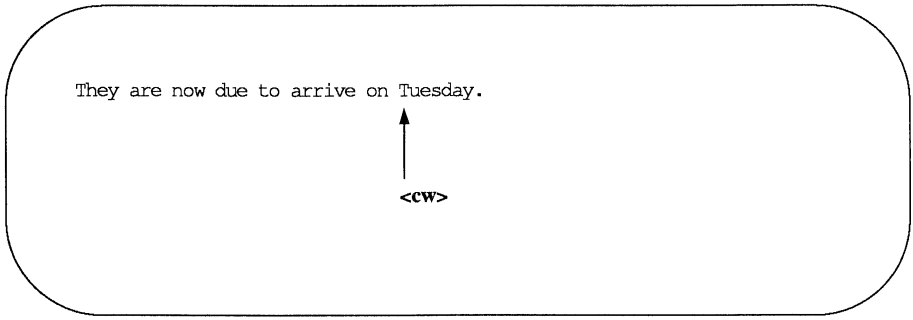
The change command can take an argument. You can replace a character, word, or an entire line with new text.

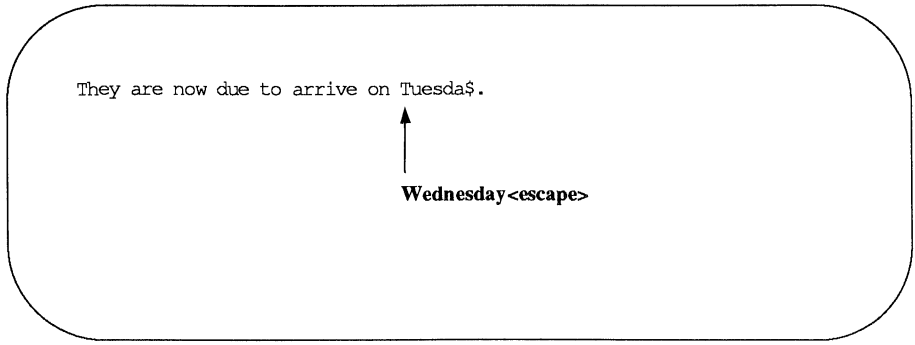
**<***n***c***x***>** Replace *n* number of text objects of type *x*, such as sentences (shown by **<)>**) and paragraphs (shown by **<}>**).

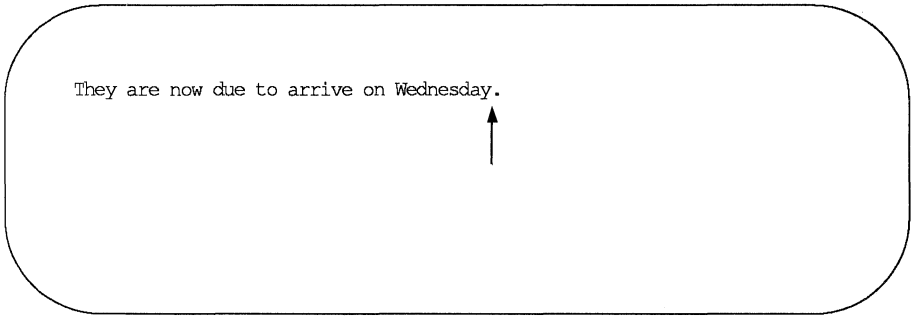
**<cw>** Replace a word or the remaining characters in a word with new text. The **vi** editor prints a $ sign to show the last character to be changed.

| | |
|---|---|
| **<*n*cw>** | Replace *n* words. |
| **<cc>** | Replace all the characters in the line. |
| **<*n*cc>** | Replace all characters in the current line and up to *n* lines of text. |
| **<C>** | Replace the remaining characters in the line, from the cursor to the end of the line. |
| **<*n*C>** | Replace 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:

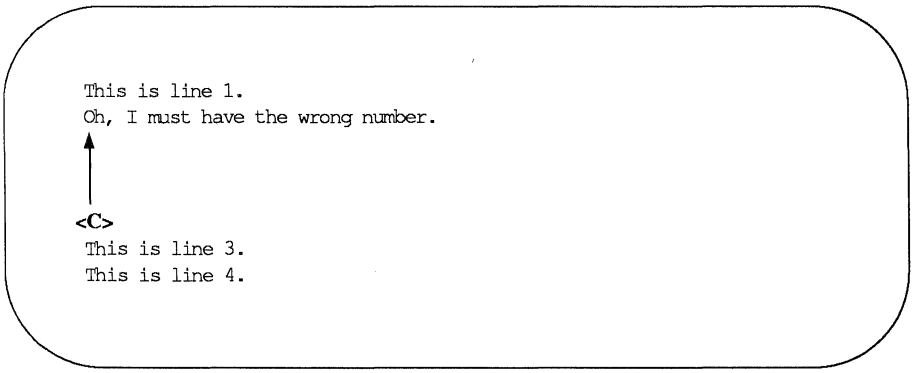


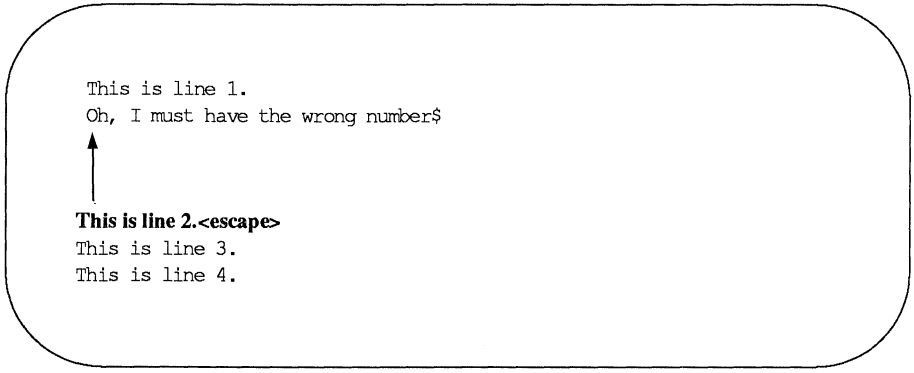



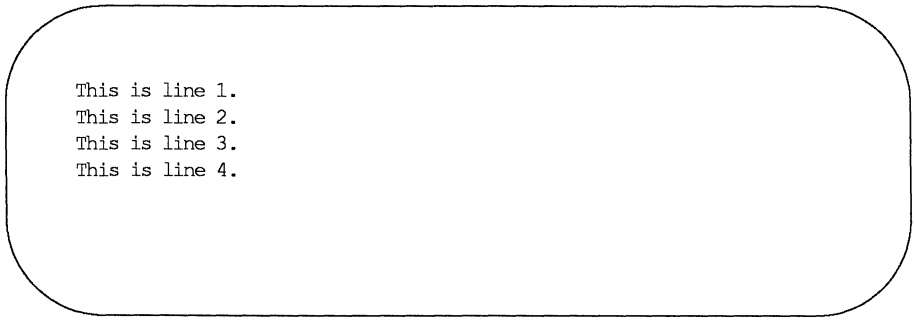



Notice that the new word (Wednesday) has more letters than the word it replaced (Tuesday). Once you have executed the change command you are in text input mode and can enter as much text as you want. The buffer will accept text until you press **<escape>**.

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.






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 **<escape>** 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.

Figure 5-10 summarizes the **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>** is pressed. |
| <s> | Delete the character the cursor is on and append text. End the append mode by pressing **<escape>**. |
| <S> | Replace all the characters in the line. |
| <cc> | Replace all the 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. |

Figure 5-10: Summary of **vi** Commands for Changing Text

# 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 will be 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 *<nx>* 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 will appear 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.

Figure 5-11 summarizes the ways you can use the yank command.

| Command | Function |
|---|---|
| <*nyx*> | 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. |

Figure 5-11: Summary of the Yank Command

Notice that this command allows you to specify the number of text objects to be yanked.

Try the following commands 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?

Figure 5-12 summarizes the cut and paste commands.

| 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. |
| **<*nyx*>** | Yank a copy of *n* number of text objects of type *x* and place them in a temporary buffer. |
| **<"x*yn*>** | Place a copy of a text object of type *n* in the register named by the letter x. |
| **<"xp>** | Place the contents of the register -x after the cursor. |

Figure 5-12: Summary of **vi** Commands for Cutting and Pasting Text

# 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` and 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 |
| <ctrl-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 will suddenly appear 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 UNIX system user sends you a message using the write command while you are editing with **vi**, the message will appear in your current window, over part of the text you are editing. To restore your text after you have read the message, you must be in command mode. (If you are in text input mode, press **<escape>** to return to command mode.) Then type **<ctrl-l>**. **vi** will erase the message and redraw the window exactly as it appeared before the message arrived.

## Making Lowercase 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.

Figure 5-13 summarizes the special commands.

| Command | Function |
|---|---|
| <> | Repeat the last command. |
| <J> | Join the line below the current line with the current line. |
| <ctrl-l> | Clear and redraw the current window. |
| <~> | Change lowercase to uppercase, or vice versa. |

Figure 5-13: Summary of Special Commands

# 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 6. 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 will also appear 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 reentering 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 **<return>** **vi** will refresh the screen and the cursor will reappear 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 **<return>**. A shell command prompt will appear 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 **<ctrl-d>** or **exit**; your screen will be refreshed with your buffer contents and the cursor will appear 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 **<ctrl-d>**.

## Writing Text to a New File: the :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<return>**

**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<return>**

A new file called **junk** will be 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***<return>**

## Finding the Line Number

To determine the line 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 **<return>**.

```
If you want to know the number
of this line, type :.=<return>


:.=
```

As soon as you press **<return>**, your command line will disappear from the bottom line and be replaced by the number of your current line in the buffer.

```
If you want to know the number
of this line, type in :.=<return>


34
```

You can move the cursor to any line in the buffer by typing : and the line number. The command line

**:*n*<return>**

means to go to the *n*th line of the buffer.

## 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<return>**

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<return>**

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

**:56r data<return>**

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***<return>**

> For each line containing *pattern*, execute the **ex** command named *command*. For example, type: **:g/helix<return>**. The line editor will print all lines that contain the pattern helix.

**:s/***pattern*/*new_words*/**<return>**

> 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<return>**

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<return>**

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 6.

Figure 5-14 summarizes the line editor commands available in **vi**.

| Command | Function |
|---|---|
| **:** | Shows that the commands that follow are line editor commands. |
| **:sh<return>** | Temporarily returns you to the shell to perform shell commands. |
| **<ctrl-d>** | Escapes the temporary shell and returns you to the current window of **vi** to continue editing. |
| *:n*<**return**> | Goes to the *n*th line of the buffer. |
| **:***x*,*y***w** *data*<**return**> | Writes lines from the number *x* through the number *y* into a new file (*data*). |
| **:$<return>** | Goes to the last line of the buffer. |
| **:.,$d<return>** | Deletes all the lines in the buffer from the current line to the last line. |
| **:r** *shell.file*<**return**> | Inserts the contents of *shell.file* after the current line of the buffer. |
| **:s/***text*/*new_words*/<**return**> | Replaces the first instance of the characters *text* on the current line with *new_words*. |
| **:s/***text*/*new_words*/**g**<**return**> | Replaces every occurrence of *text* on the current line with *new_words*. |
| **:g/***text*/**s//***new_words*/**g**<**return**> | Replaces every occurrence of *text* in the file with *new_words*. |

Figure 5-14: Summary of Line Editor Commands

# 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<return>** | Write the contents of the **vi** buffer to the UNIX file currently being edited and quit **vi**. |
| **:w** *filename<return>* **:q<return>** | Write the temporary buffer to a new file named *filename* and quit **vi**. |
| **:w!** *filename<return>* **:q<return>** | Overwrite an existing file called *filename* with the contents of the buffer and quit **vi**. |
| **:q!<return>** | Quit **vi** without writing the buffer to a file, and discard all changes made to the buffer. |
| **:q<return>** | Quit **vi** without writing the buffer to a UNIX file. This works only if you have made no changes to the buffer; otherwise **vi** will warn you that you must either save the buffer or use the **:q!<return** 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<return>**

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<return>**

After you write to the new file, leave **vi**. Type:

**:q<return>**

If you try to write to an existing file, you will receive a warning. For example, if you try to write to a file called **johnson**, the system will respond 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<return>**

Your new file will overwrite the existing one.

If you edit a file called **memo**, make some changes to it, and then decide you don't 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!<return>**

Figure 5-15 summarizes the quit commands.

| Command | Function |
|---|---|
| **<ZZ>** | Write the file and quit **vi**. |
| **:wq<return>** | Write the file and quit **vi**. |
| **:w** *filename***<return>** **:q<return>** | Write the editing buffer to a new file (*filename*) and quit **vi**. |
| **:w!** *filename***<return>** **:q<return>** | Overwrite an existing file (*filename*) with the contents of the editing buffer and quit **vi**. |
| **:q!<return>** | Quit **vi** without writing buffer to a file. |
| **:q<return>** | Quit **vi** without writing the buffer to a file. |

Figure 5-15: Summary of the Quit Commands

# Special Options For vi

The **vi** command has some special options. It allows you to:

- recover a file lost by an interrupt to the UNIX 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 will exit the **vi** command without writing the text in the buffer back to its file. However, the UNIX system will store a copy of the buffer for you. When you log back in to the UNIX system you will be able to restore the file with the **–r** option for the **vi** command. Type

**vi –r** *filename*<**return**>

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**. **vi** 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** *file1 file2*<**return**>

**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**<**return**>

The system response to the **:w** <**return**> command will be 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**<**return**>

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**.

Figure 5-16 summarizes the special options for **vi**.

| Option | Function |
|---|---|
| **vi** *file1 file2 file3*<**return**> | Enter three files (*file1*, *file2*, and *file3*) into the **vi** buffer to be edited. |
| **:w**<**return**><br>**:n**<**return**> | Write the current file and call the next file into the buffer. |
| **vi –r** *file1*<return> | Restore the changes made to *file1*. |

Figure 5-16: Summary of Special Options for **vi**

# 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*<return>**

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 terminal's system name. Type:

**TERM=***terminal_name***<return>**

1-2. Enter the **vi** command for a file called **exer1**:

**vi exer1<return>**

Then use the append command (**<a>**) to enter the following text in your file:

**This is an exercise!<return>**
**Up, down<return>**
**left, right,<return>**
**build your terminal's<return>**
**muscles bit by bit<escape>**

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<escape>**

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

**<return>**. To leave text input mode, press **<escape>**.

1-7. Type:

**<ZZ>**

1-8. Type:

**vi exer1<return>**

Notice the system response:

```
"exer1" 7 lines, 102 characters
```

## Exercise 2

2-1. Type:

**vi exer2<return>**
**<a>1<return>**
**2<return>**
**3<return>**
.
.
.
**48<return>**
**49<return>**
**50<escape>**

2-2. Type:

**<ctrl-f>**
**<ctrl-b>**
**<ctrl-u>**
**<ctrl-d>**

Notice the line numbers as the screen changes.

2-3. Type:

**<G>**
**<o>**
**123456789 123456789<escape>**
**<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<return>**

3-2. Type:

**<a> Append text <return>**
**Insert text<return>**
**a computer's <return>**
**job is boring.<escape>**

3-3. Type:

**<O>**
**financial statement and<escape>**

3-4. Type:

**<3G>**
**<i>Delete text<return><escape>**

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.
**<return>** creates the new line.
Add the sentence: **But, it is an exciting machine.**
**<escape>** leaves append mode.

3-7. Type:

**<1G>**
**/text**
**<i>some<space bar><escape>**

3-8. **<ZZ>** will write the buffer to **exer3** and return you to the shell.

## Exercise 4

4-1. Type:

**vi exer4<return>**
**<a> When in the course of human events<return>**
**there are many repetitive, boring<return>**
**chores, then one ought to get a<return>**
**robot to perform those chores.<escape>**

4-2. Type:

**<2G>**
**<A> tedious and unsavory<8backspace><return>**
**<escape>**

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<escape>**
**<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<return>**

# Exercise 5

5-1. Type:

**vi exer2<return>**
**<8G>**
**<cc> END OF FILE <escape>**

5-2. Type:

**<1G>**
**<8"zyy>**
**<G>**
**<"zp>**

5-3. Type:

**<8G>**
**<cc> 8 is great<escape>**

5-4. Type:

**<G>**
**<2w>**
**<cw>**
**EXERCISE<escape>**
**<2b>**
**<cw>**
**TO<escape>**

# Exercise 6

6-1. Type:

**vi exer6<return>**
**<a>** (append several lines of text)
**<escape>**

Turn off the terminal.

Turn on the terminal.
Log in on your UNIX system. Type:

**vi –r exer6<return>**
**:wq<return>**

6-2. Type:

**vi exer1 exer2<return>**
**:w<return>**
**:n<return>**

**:w junk<return>**
**<ZZ>**

6-3. Type:

**vi exer*<return>**

(Response:)
`8 files to edit` (**vi** calls all files with names that begin with **exer**.)

**<ZZ>**
**<ZZ>**

6-4. Type:

**view exer4<return>**
**<ctrl-f>**
**<ctrl-d>**
**<ctrl-b>**
**<ctrl-u>**
**:q<return>**

# The ed 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, "The Bourne Shell Tutorial").

**ed** is a line editor; during editing sessions it is always pointing to 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** will perform each command you issue on the current line. In addition to letting you change, delete, or add text on one or more lines, **ed** allows 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 each 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 command, to move the contents of the buffer into the file.

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

- 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 information

# Suggestions for Using this Tutorial

The commands discussed in each section are reviewed at the end of that section. A summary of all **ed** commands introduced in this chapter is found in a quick reference guide immediately following this chapter.

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.

The notation conventions used in this chapter are those used throughout this guide. They are described in the Preface.

# Getting Started

The best way to learn **ed** is to log in to the UNIX system and try the examples as you read this tutorial. 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**

## How to Enter ed

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

**ed** *filename*<**return**>

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<return>
?new-file
```

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

```
% ed old-file<return>
235
```

## How to Create 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**<return>**:

**a<return>**

Now you are in input mode; any characters you type from this point will be 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 give **ed** other commands.

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

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

Notice that **ed** does not respond to the period; it 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 **<return>** to regain command mode. For example, if you added unwanted characters or lines to your text, you can delete them once you have returned to command mode.

## How to Display Text

To display a line of a file, type **p** (for print) on a line by itself. The **p** command prints the current line, that is, 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<return>
? try-me
a<return>
This is the first line of text.<return>
This is the second line,<return>
and this is the third line.<return>
.<return>
p<return>
and this is the third line.
```

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

```
2p<return>
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<return>
```

You can even print the whole 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, the **ed** symbol for the address of the last line. (These conventions are discussed in detail in the section "Line Addressing.")

```
1,$p<return>
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 and then try the **p** command without quitting text input mode. Then quit text input mode and print the entire file.

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

What did you get? The next section will explain how to delete the unwanted line.

## How to Delete 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 this:

```
p<return>
p
d<return>
1,$p<return>
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 this:

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

## How to Move Up or Down in the File

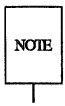
To display the line below the current line, press **<return>** while in command mode. If there is no line below the current line, **ed** responds with a ? and continues to treat the last line of the file as the current line. To display the line above the current line, press the minus key (–). The following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

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

Notice that by typing **-<return>**, 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.

## How to Save the Buffer Contents in a File

As we discussed earlier, during an editing session, the system holds your text in a temporary storage area called a buffer. When you have finished editing, you can save your work by writing it from the temporary buffer to a permanent file in the computer's memory. By writing to a file, you are simply 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.




It is a good idea to write the buffer text into your file frequently. If an interrupt occurs (such as an accidental loss of power to 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 **<return>**. If you have just created new text, **ed** creates a file for it with the name you specified when you entered the editor. If you have edited an existing file, the **w** command writes the contents of the buffer to that file by default.

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** will not warn you about an existing file; it will simply overwrite 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<return>
? try-me
a<return>
This is the first line of text.<return>
This is the second line,<return>
and this is the third line.<return>
.
w stuff <return>
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.

## How to Quit 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<return>
85
q<return>
%
```

The system responds with a shell prompt. At this point the editing buffer vanishes. If you have not executed 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 in this way. 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<return>
?
w<return>
85
q<return>
%
```

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 wiped out.

You now know the basic commands needed to create and edit a file using **ed**. Figure 6-1 summarizes these commands.

| Command | Function |
|---|---|
| **ed** *file* | enter **ed** to edit *file* |
| **a** | append text after the current line |
| **.** | quit text input mode and return to **ed** command mode. |
| **p** | print text on your terminal |
| **d** | delete text |
| **<return>** | display the next line in the buffer (literally, carriage return) |
| **+** | display the next line in the buffer |
| **–** | display the previous line in the buffer |
| **w** | write the contents of the buffer to the file |
| **q** | quit **ed** and return to the shell |

Figure 6-1: Summary of **ed** Editor Commands

# Exercise 1

Answers for all the exercises in this chapter are found 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*]**<return>**

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

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

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

*argument*
: The *arguments* to a *command* 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*<**return**>

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:

- by 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)
- by entering special symbols for the current line, last line, or a span of lines
- by adding or subtracting lines from the current line
- by 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, such as a word.)

## Numerical Addresses

**ed** gives a numerical address to each line in the buffer. The first line of the buffer is 1, the second line is 2, and so on, for each line in the buffer. Any line can be accessed by **ed** with its line address number. To see how line numbers address a line, enter **ed** with the file **try-me** and type a number.

```
% ed try-me<return>
110
1<return>
This is the first line of text.
3<return>
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.

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

# Symbolic Addresses

## Symbolic Address of 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 simply by typing a period (.) and pressing **<return>**.

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

```
% ed try-me<return>
110
.<return>
This is the fourth line.
```

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

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

**.=<return>**

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

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

## Symbolic Address of 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<return>
110
.<return>
This is the fourth line.
$<return>
This is the fourth line.
```

## Symbolic Address of the Set of All Lines

When used as an address, a comma (,) refers to all the 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<return>
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line.
```

## Symbolic Address of a Set of Lines

The semicolon (;) 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<return>
This is the second line,
;p<return>
This is the second line,
and this is the third line.
This is the fourth line.
```

## Adding or Subtracting from the Current Line

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 will be saved:

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

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

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

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

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. "Other Useful Commands and Information," at the end of this chapter, explains how to get a help message that describes the error.

## Character String Addresses

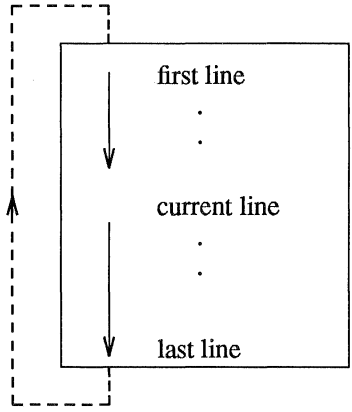
You can search forward or backward in the file for a line containing a particular character string. To do so, specify a string, preceded by a delimiter.

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*

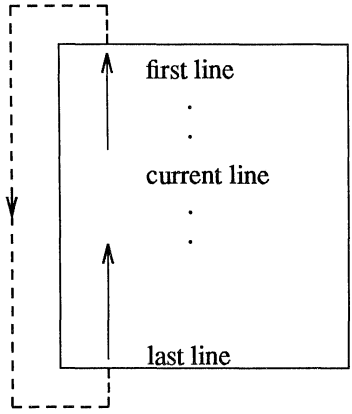
When you specify a pattern preceded by a / (slash), **ed** begins at the current line and searches forward (down through subsequent lines in the buffer) for the next line containing the *pattern*. When the search reaches the last line of the buffer, **ed** wraps around to the beginning of the file and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

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




Another useful delimiter is ?. If you specify a pattern preceded by a ?, (*?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 will wrap around and continue 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<return>
140
.<return>
ten
?first<return>
This is the first line of text.
/fourth<return>
This is the fourth line.
/junk<return>
?
```

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

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*, as usual. After **ed** has printed the first occurrence, it waits for another command. Type / and press **<return>**; **ed** will continue 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**:

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

Notice that after **ed** has found all occurrences of the *pattern* between the line where you requested a search and the end of the 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, such as *address1* through *address2*, or you can specify a global search for all lines containing a specified pattern.

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

**2,7p<return>**

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

```
2,7p<return>
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<return>
This is the fourth line
-2,+3p<return>
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 general format for these commands is:

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

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

```
g/line/p<return>
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<return>
five
six
seven
eight
nine
ten
```

Notice the function of the **v** command: it finds all the lines that do not contain the word specified in the command 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<return>
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. Figure 6-2 summarizes the symbols and commands available for addressing lines.

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

Figure 6-2: Summary of Line Addressing

# 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<return>**

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

**?<return>**

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

**ed** provides two commands for displaying lines of text in the editing buffer: **p** and **n**.

## Displaying Text Alone: the p Command

You have already used the **p** command in several examples. You are probably now familiar with its general format:

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

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

Experiment with the line addresses shown in Figure 6-3 on a file in your home directory. Try the **p** command with each address and see if **ed** responds as described in the figure.

| Specify this Address | Check for this Response |
|---|---|
| **1,$p<return>** | **ed** should display the entire file on your terminal. |
| **–5p<return>** | **ed** should move backward five lines from the current line and display the line found there. |
| **+2p<return>** | **ed** should move forward two lines from the current line and display the line found there. |
| **1,/*x*/p<return>** | **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* between slashes so that **ed** can distinguish between the search pattern address (*x*) and the **ed** command (**p**). |

Figure 6-3: Sample Addresses for Displaying Text

# Displaying Text with Line Addresses: the n Command

The **n** command displays text and precedes each line with its numerical line address. It is helpful when you are deleting, creating, or changing lines. The general command line format for **n** is the same as that for **p**.

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

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<return>
140
1,$n<return>
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
```

Figure 6-4 summarizes the **ed** commands for displaying text.

| Command | Function |
|---|---|
| **p** | displays specified lines of text in the editing buffer on your terminal |
| **n** | displays specified lines of text in the editing buffer with numerical line addresses on your terminal |

Figure 6-4: Summary of Commands for Displaying Text

# Creating Text

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

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

## Appending Text: the a Command

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

[*address1*]**a<return>**

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

Notice that after you append the two new lines, the line 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<return> appends text after the current line**

**$a<return> appends text after the last line of the file**

**0a<return> 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<return>
?lines
a<return>
This is the current line.<return>
.<return>
p<return>
This is the current line.
.a<return>
This line is after the current line.<return>
.<return>
-1,.p<return>
This is the current line.
This line is after the current line.
```

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

```
0a<return>
This is the first line now.<return>
This is the second line now.<return>
The line numbers change<return>
as lines are added.<return>
.<return>
1,4n<return>
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 previous 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: the 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 that for **a**.

[*address1*]**i<return>**

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<return>
?insert
a<return>
Line 1<return>
Line 2<return>
Line 3<return>
Line 4<return>
.<return>
w<return>
69
```

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

```
2i<return>
This is the new line 2.<return>
.<return>
1,$n<return>
1       Line 1
2       This is the new line 2.
3       Line 2
4       Line 3
5       Line 4
1i<return>
This is the beginning.<return>
.<return>
1,$n<return>
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, as follows:

- **.i<return>**
- **$i<return>**

## Changing Text: the 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<return>**

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<return>
1       line 1
2       line 2
3       line 3
4       line 4
5       line 5
1,4c<return>
Change line 1<return>
and lines 2 through 4<return>
.<return>
1,$n<return>
1       change line 1
2       and lines 2 through 4
3       line 5
```

Now experiment with **c** and try to change the current line:

```
.<return>
line 5
c<return>
This is the new line 5.
.<return>
.<return>
This is the new line 5.
```

If you are not sure whether 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**.

Figure 6-5 summarizes the **ed** commands for creating text.

| Command | Function |
|---|---|
| **a** | append text after the specified line in the buffer |
| **i** | insert text before the specified line in the buffer |
| **c** | change the text on the specified line(s) to new text |
| **.** | quit text input mode and return to **ed** command mode |

Figure 6-5: Summary of Commands for Creating Text

# 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<return>**

Insert above the current line:

**or<return>**
**Naperville<return>**

Insert before the last line:

**hotels in<return>**

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<return>**

Display lines 1 through 3.

3-4. After you have completed 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 text input mode: **<#>** (the pound sign) deletes a character and **<@>** (the at sign) 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: the 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<return>**

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<return>
1               1 horse
2               2 chickens
3               3 ham tacos
4               4 cans of mustard
5               5 bales of hay
2,4d<return>
1,$n<return>
1               1 horse
2               5 bales of hay
```

How can you delete only the last line of a file? Using a symbolic line address makes this easy:

**$d<return>**

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 . is used to delete it.

```
a<return>
Last line of text<return>
1,$p<return>
.<return>
p<return>
1,$p
.d<return>
p<return>
Last line of text.
```

Before experimenting with the delete command, you may first want to learn about the undo command, **u**.

## Undoing the Previous Command: the 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<return>**

One purpose for which the **u** command is useful is to restore text you have mistakenly deleted. If you delete all the lines in a file and then type **p, ed** will respond with a ? since there are no more lines in the file. Use the **u** command to restore them.

```
1,$p<return>
This is the first line.
This is the middle line.
This is the last line.
1,$d<return>
p<return>
?
u<return>
p<return>
This is the last line.
```

Now experiment with **u**: use it to undo the append command.

```
.<return>
This is the only line of text
a<return>
Add this line<return>
.<return>
1,$p<return>
This is the only line of text
Add this line
u<return>
1,$p<return>
This is the only line of text
```


**u** cannot be used to undo the write command (**w**) or the quit command (**q**). However, **u** can undo an undo command (**u**).

# How to Delete 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 @ sign key kills the current line. The # sign key backs up over one character on the current line so you can retype it, thus effectively 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 have reassigned these functions, you must use the keys you chose while working in **ed**; the default keys (@ and #) will no longer work.

## Escaping the Delete Function

You may want to include an @ sign or a # sign as a character 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<return>
leave San Francisco \@ 20:15 on flight \#347 <return>
.<return>
p<return>
leave San Francisco @ 20:15 on flight #347
```

Figure 6-6 summarizes the **ed** commands and shell commands for deleting text in **ed**.

| Command | Function |
|---|---|
| In command mode: | |
| **<d>** | delete one or more lines of text |
| **<u>** | undo the previous command |
| **<@>** | delete the current command line |
| In text input mode: | |
| **<@>** | delete the current line |
| **<#>** or **<backspace>** | delete the last character typed in |

Figure 6-6: Summary of Commands for Deleting Text

# Substituting Text

You can modify 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*]**<return>**

Each component of the command line is described below.

*address1*,*address2*
: 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**
: The substitute command

/*old_text*
: The argument specifying the text to be replaced is usually delimited by slashes, but can be delimited by other characters such as a ? or a period. It consists of the words or characters to be replaced. The command will replace the first occurrence of these characters that it finds in the text.

/*new_text*
: The argument specifying 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*
: Any one of the following four commands:

**g**
: Change all occurrences of *old_text* on the specified lines.

**l**
: Display the last line of substituted text, including nonprinting characters. (See the last section of this chapter, "Other Useful Commands and Information.")

**n**
: Display the last line of the substituted text preceded by its numerical line address.

**p**
: Display 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***/<return>**

The next example contains a typing error. While the line that contains it is still the current line, you make a substitution to correct it. The old text is the **ai** of **airor** and the new text is **er**.

```
a<return>
In the beginning, I made an airor.
.<return>
.p<return>
In the beginning, I made an airor.
s/ai/er<return>
```

Notice that **ed** gives no response to the substitute command. To verify that the command has succeeded in this case, you either have to display the line with **p** or **n**, or include **p** or **n** as part of the substitute command line. In the following example, **n** is used to verify that the word **file** has been substituted for the word **toad**.

```
.p<return>
This is a test toad
s/toad/file/n<return>
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<return>
This is a test file
s/file/frog<return>
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, as follows:

[*address1*]s/*old_text*/*new_text*/**<return>**

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<return>
This is a pest toad
testing testing
come in toad
.<return>
come in toad
1s/pest/test<return>
This is a test toad
```

As you can see, **ed** printed the new line automatically after the change was made, because the last delimiter was omitted.

## Substituting on a Range of Lines

You can make a substitution on a range of lines by specifying the first address (*address1*) through the last address (*address2*).

[*address1*,*address2*]s/*old_text*/*new_text*/**<return>**

If **ed** does not find the pattern to be replaced on a line, no changes are made to that line.

In the following example, all the lines in the file are addressed for the substitute command. However, only the lines that contain the string **es** (the *old_text* argument) are changed.

```
1,$p<return>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
1,$s/es/ES/n<return>
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<return>
1       This is a tESt toad
2       tESting testing
3       come in toad
4       tESting 1, 2, 3
```

Notice that only the first occurrence of **es** (on line 2) has been changed. To change every occurrence of a pattern, use the **g** command, described in the next section.

## Global 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 es in the last example. If you are following along, doing the examples as you read this, remember you can use **u** to undo the last substitute command.

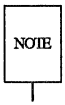
```
u<return>
1,$p<return>
This is a test toad
testing, testing
come in toad
testing 1, 2, 3
1,$s/es/ES/g<return>
1,$p<return>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Another method is to use a global search pattern as an address instead of the range of lines specified by **1,$**.

```
1,$p<return>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/test/s/es/ES/g<return>
1,$p<return>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

If the global search pattern is unique and matches the argument *old_text* (text to be replaced), you can use an **ed** shortcut: specify the pattern once as the global search address and do not repeat it as an *old_text* argument. **ed** will remember the pattern from the search address and use it again as the pattern to be replaced.

**g/***old_text***/s//***new_text***/g<return>**




Whenever you use this shortcut, be sure to include two slashes (//) after the **s**.

```
1,$p<return>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/es/s//ES/g<return>
1,$p<return>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Experiment with other search pattern addresses:

**/***pattern***<return>**
**?***pattern***<return>**
**v/***pattern***<return>**

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<return>
This is a test toad
come out toad
```

Notice that the line `This is a test toad` was also printed, even though no substitution was made on it. When the last delimiter is omitted, all lines found with the search address are printed, regardless of whether or not substitutions have been made on them.

Now search for lines that do contain the pattern `testing` with the **g** command.

```
g/testing/s//jumping<return>
jumping testing
jumping 1, 2, 3
```

Notice that this command makes substitutions only for the first occurrence of the *pattern* (`testing`) in each line. Once again, the lines are displayed on your terminal because the last delimiter has been omitted.

# 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
```

and substitute

**found**

using the **?** as the delimiter. Did it work?

4-4. Search forward in the file for

```
no
```

and substitute

**NO**

for it. 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 $<return>**

What happened?

# Special Characters

If you try to substitute the $ sign in the line

```
I lost my $ in Las Vegas
```

you will find that instead of replacing the $, the new text is placed at the end of the line. 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:

| | |
|---|---|
| . | Match any one character. |
| * | Match zero or more occurrences of the preceding character. |
| .* | Match zero or more occurrences of any character following the period. |
| ^ | Match the beginning of the line. |
| $ | Match the end of the line. |
| \ | Take away the special meaning of the special character that follows. |
| & | Repeat the old text to be replaced in the new text of the replacement pattern. |
| [...] | Match the first occurrence of a character in the brackets. |
| [^...] | Match 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<return>
rat
cat
turtle
cow
goat
g/.at<return>
rat
cat
goat
```

Notice that 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, suppose 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<return>
brrroke
s/br*/br<return>
broke
```

Notice that 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<return>
brrroke
s/r*/r<return>
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<return>
Toads are slimy, cold creatures
s/are.*/are wonderful and warm<return>
Toads are wonderful and warm
```

The .* can also replace all characters between two patterns.

```
p<return>
Toads are slimy, cold creatures
s/are.*cre/are wonderful and warm cre<return>
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,$p<return>
creatures great and small
things wise and wonderful
things bright and beautiful
1,$s/^/all /<return>
1,$p<return>
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,$p<return>
I love
I need
I use
The IRS wants my
1,$s/$/ money.<return>
1,$p<return>
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/<return>
1,$p<return>
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,$p<return>
I love money
I need money
I use money
The IRS wants my money
1,$s/$/./<return>
1,$p/<return>
I love money.
I need money.
I use money.
The IRS wants my money.
```

Because . is not matching a 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<return>
Way to go.  Wow!
s/\./!<return>
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<return>
This chapter explains
how to use the \.
s/\\backslash<return>
how to use the backslash.
```

If you want to add text without changing the rest of the line, the & provides a useful shortcut. The & (ampersand) repeats the old text in the replacement pattern, so you do not have to type the pattern twice. For example:

```
p<return>
The neanderthal skeletal remains
s/thal/& man's/<return>
p<return>
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 the % sign. The % 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,$n<return>
 1       I love money
 2       I need food
 3       I use money
 4       The IRS wants my money
 1s/money/gold<return>
 I love gold
 3s//%%<return>
I use gold
 1,$n<return>
 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. The % sign 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,$p<return>
Monday          33,000
Tuesday         75,000
Wednesday       88,000
Thursday        62,000
1,$s/[6789]/4<return>
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,$p<return>
Mr Arthur Middleton
Mr Matt Lewis
Ms Anna Kelley
Ms M. L. Hodel
1,$s/M[rs] //<return>
1,$p<return>
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,$p<return>
grade  A   Computer Science
grade  B   Robot Design
grade  A   Boolean Algebra
grade  D   Jogging
grade  C   Tennis
1,$s/grade [^AB]/grade A<return>
1,$p<return>
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

**1,$s/[^AB]/A<return>**

you would have changed the g in the word grade to A. Try it.

Experiment with these special characters. Find out what happens (or does not happen) if you use them in different combinations.

Figure 6-7 summarizes the special characters for search or substitute patterns.

| Command | Function |
|---|---|
| . | Match any one character in a search or substitute pattern. |
| * | Match zero or more occurrences of the preceding character in a search or substitute pattern. |
| .* | Match zero or more occurrences of any characters following the period. |
| ^ | Match the beginning of the line in the substitute pattern to be replaced or in a search pattern. |
| $ | Match the end of the line in the substitute pattern to be replaced. |
| \ | Take away the special meaning of the special character that follows in the substitute or search pattern. |
| & | Repeat the old text to be replaced in the new text replacement pattern. |
| % | Match the last replacement pattern. |
| [...] | Match the first occurrence of a character in the brackets. |
| [^...] | Match the first occurrence of a character that is NOT in the brackets. |

Figure 6-7: Summary of Special Characters

# 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/<return>**

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

**10202031020**

without repeating the old digits in the replacement pattern?

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** | move lines of text |
| **t** | copy lines of text |
| **j** | join contiguous lines of text |
| **w** | write lines of text to a file |
| **r** | read in the contents of a file |

## Move 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*]**<return>**

The components of this command line include:

*address1*,*address2*
: 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**
: The move command.

*address3*
: Place the text after this line.

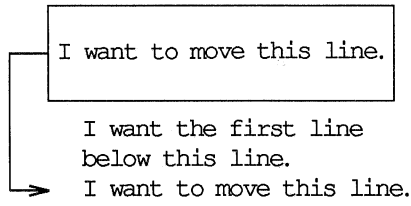
Try the following example to see how the command works. Create a file that contains these three lines of text:

**I want to move this line.**
**I want the first line**
**below this line.**

Type:

**1m3<return>**

**ed** will move line 1 below line 3.




The next screen shows how this will appear on your terminal:

```
1,$p<return>
I want to move this line.
I want the first line
below this line.
1m3<return>
1,$p<return>
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<return>
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<return>
64              Move the block of text
65              below this line.
8,12m65<return>
59,65n<return>
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<return>**

When *address3* is 0, the lines are placed at the beginning of the file.

# Copy 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*]**<return>**

*address1*,*address2*
: 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**
: The copy command.

*address3*
: Place 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<return>
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<return>
30        A chemical fire in the lab requires that you:
6,8t30<return>
30,$n<return>
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<return>
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<return>**

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<return>
Now is the time to join
the team.
p<return>
the team.
1p<return>
Now is the time to join
j<return>
p<return>
Now is the time to jointhe team.
```

Notice that 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.

## Write 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*]**<return>**

*address1*,*address2*
: 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**
: The write command.

*filename*
: 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<return>
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<return>
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.

## Problems

The **w** command overwrites preexisting 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 "Other Useful Commands and Information," 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).

## Read 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*<**return**>

*address1* 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** The read command.

*filename* 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,$n<return>
1                       March 17, 1986
2               Dear Michael,
3               Are you free later today?
4               Hope to see you there.
3r memo<return>
87
3,$n<return>
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.

Figure 6-8 summarizes the **ed** commands for moving text.

| Command | Function |
|---|---|
| **m** | move lines of text |
| **t** | copy lines of text |
| **j** | join contiguous lines |
| **w** | write text into a new file |
| **r** | read in text from another file |

Figure 6-8: Summary of **ed** Commands for Moving Text

# 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. Say 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?

# Other Useful Commands and Information

There are four other commands and a special file that will be useful to you 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 reason for the most recent ?.

**H** Places **ed** into help mode so that a short error message is displayed 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<return>
?
h<return>
warning: expecting 'w'
```

The ? is also displayed 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<return>
? newfile
h<return>
cannot open input file
```

This message means one of two things: either there is no file called *newfile* or there is such a file but **ed** is not allowed to read it.

As explained earlier, the **H** command responds to the ? and then turns on the help mode of **ed**, so that **ed** gives you a diagnostic explanation every time the ? is subsequently displayed. To turn off help mode, type **H** again. The next screen shows **H** being used to turn on help mode. Sample error messages are also displayed in response to some common mistakes:

```
% ed newfile<return>
e newfile<return>
?newfile
H<return>
cannot open input file
/hello<return>
?
illegal suffix
1,22p<return>
?
line out of range
a<return>
I am appending this line to the buffer.
.<return>
s/$ tea party<return>
?
illegal or missing delimiter
,$s/$/ tea party<return>
?
unknown command
H<return>
q<return>
?
h<return>
warning: expecting 'w'
```

These are some of the most common error messages that you may encounter during editing sessions:

`illegal suffix`
: **ed** cannot find an occurrence of the search pattern **hello** because the buffer is empty.

`line out of range`
: **ed** cannot print any lines because the buffer is empty or the line specified is not in the buffer.

A 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.

## Display Nonprinting Characters

If you are typing a tab character, the terminal will normally display up to eight spaces covering the space up to the next tab setting.

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<return>**

The components of this command line are:

*address1*,*address2*
The range of lines to be displayed. If no address is given, the current line will be displayed. If only *address1* is given, only that line will be displayed.

**l**
The command that displays the nonprinting characters along with the text.

The **l** command denotes tabs with a **>** (greater than) character. To type control characters, hold down the control key and press the appropriate alphabetic key. The key that sounds the bell is **<ctrl-g>**. It is displayed as \07 which is the octal representation (the computer's code) for **<ctrl-g>**.

Type in two lines of text that contain a **<ctrl-g>** and a tab. Then use the **l** command to display the lines of text on your terminal.

```
a<return>
Add a <ctrl-g> to this line.<return>
Add a <tab> (tab) to this line.<return>
.<return>
1,2l<return>
Add a \07 (control-g) to this line.<return>
Add a > (tab) to this line.<return>
```

Did the bell sound when you typed **<ctrl-g>**?

## The Current File Name

In a long editing session, you may forget the file name. The **f** command will remind 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<return>**

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<return>
323
f<return>
oldfile
```

To associate the contents of the editing buffer with a new file name use this general format:

**f newfile<return>**

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* and then changing its name to *newfile*. A line of text is added to the buffer and then the write and quit commands are issued.

```
ed oldfile<return>
323
f<return>
oldfile
f newfile<return>
newfile
a<return>
Add a line of text.<return>
.<return>
w<return>
343
q<return>
```

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.

## Escape to the Shell

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<return>
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 program's response to your command will appear as the command is running. When the command has finished executing, the ! will appear alone on a line. This means that that you are back in the editor at the current line.

For example, if you want to return to the shell to find out the correct date, type **!** and the shell command **date.**

```
p<return>
This is the current line
! date<return>
Tue  Apr 1  14:24:22  EST  1986
!
p<return>
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 is displayed, 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 UNIX system tries to save the contents of the editing buffer in a special file named **ed.hup.** Later you can retrieve your text from this file in one of two ways. First, you can use a shell command to move **ed.hup** to another file name, such as the name the file had while you were editing it (before the interrupt). Second, you can enter **ed** and use the **f** command to rename the contents of the buffer. An example of the second method is shown in the following screen:

```
ed ed.hup<return>
928
f myfile<return>
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 now are 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.

Figure 6-9 summarizes the functions of the commands introduced in this section.

| Command | Function |
|---|---|
| **h** | Displays a short error message for the preceding diagnostic ?. |
| **H** | Turn on help mode. An error message will be given with each diagnostic *?*. The second **H** turns off help mode. |
| **l** | Display nonprinting characters in the text. |
| **f** | Display the current file name. |
| **f** *newfile* | Change the current file name associated with the editing buffer to *newfile*. |
| *!cmd* | Temporarily escape to the shell to execute a shell command *cmd*. |
| **ed.hup** | The editing buffer is saved in **ed.hup** if the terminal is hung up before a write command. |

Figure 6-9: Summary of Other Useful Commands

# Exercise 7

7-1. Create a new file called **newfile1**. Access **ed** and change the file's 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**? 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<return>
? junk
a<return>
Hello world.<return>
.<return>
w<return>
12
q<return>
%
```

1-2.

```
% ed junk<return>
12
1,$p<return>
Hello world.<return>
q<return>
%
```

The system did not respond with the warning question mark because you did not make any changes to the buffer.

1-3.

```
% ed junk<return>
12
a<return>
Wendy's horse came through the window.<return>
.<return>
1,$p<return>
Hello world.
Wendy's horse came through the window.
q<return>
?
w stuff<return>
60
q<return>
%
```

# Exercise 2

2-1.

```
% ed towns<return>
? towns
a<return>
My kind of town is<return>
Chicago<return>
Like being no where at all in<return>
Toledo<return>
I lost those little town blues in<return>
New York<return>
I lost my heart in<return>
San Francisco<return>
I lost $$ in<return>
Las Vegas<return>
.<return>
w<return>
164
```

2-2.

```
3<return>
Like being no where at all in
```

2-3.

```
-2,+3p<return>
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.

```
.=<return>
6
6<return>
New York
```

2-5.

```
%<return>
Las Vegas
```

2-6.

```
?town<return>
I lost those little town blues in
?<return>
My kind of town is
```

2-7.

```
g/in<return>
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<return>
Chicago
Toledo
New York
San Francisco
Las Vegas
```

# Exercise 3

3-1.

```
% ed ex3<return>
?ex3
i<return>
?
q<return>
```

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<return>
164
.n<return>
10              Las Vegas
3i<return>
Illinois<return>
.<return>
.i<return>
or<return>
Naperville<return>
.<return>
$i<return>
hotels in<return>
1,$n<return>
 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<return>
1  My kind of town is
2  Chicago
3  or
4  Naperville
5  Illinois
2,5c<return>
London<return>
.<return>
1,3n<return>
1  My kind of town is
2  London
3  Like being no where at all
```

3-4.

```
.<return>
Like being no where at all
/Tol<return>
Toledo
c<return>
Peoria<return>
.<return>
.<return>
Peoria
```

3-5.

```
.<return>
/New Y/c<return>
Iron City<return>
.<return>
.<return>
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<return>
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 line

```
I lost those little town blues in
```

was not printed because it was NOT addressed by the v command.

4-2.

```
.<return>
Las Vegas
s?Las Vegas?Toledo<return>
Toledo
```

4-3.

```
?lost?s??found<return>
I found $$ in
```

4-4.

```
/no?s??NO<return>
?
/no/s//NO<return>
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<return>
? file1
a<return>
A Computer Science<return>
D Jogging<return>
C Tennis<return>
.<return>
1,$s/[^AB]/A/<return>
1,$p<return>
A Computer Science
A Jogging
A Tennis
u<return>
```

```
1,$s/^[^AB]/A<return>
1,$p<return>
A Computer Science
A Jogging
A Tennis
```

5-2.

```
2i<return>
These are not really my grades.<return>
1,$p<return>
A  Computer Science
These are not really my grades.
A  Tennis
A  Jogging
/^[^A]<return>
These are not really my grades
?^[T]<return>
These are not really my grades
```

5-3.

```
1,$p<return>
I love money
I need money
The IRS wants my money
g/^I/s/I.*m /It's my m<return>
It's my money
It's my money
```

```
/s/money/gold<return>
It's my gold
2,$s//%<return>
The IRS wants my gold
```

5-4.

```
s/10202/&0<return>
10202031020
```

5-5.

```
a<return>
*.\& %^**<return>
.<return>
s/*/a<return>
a . \ & % ^ *
s/*/b<return>
a . \ & % ^ b
```

Because there were no preceding characters, * substituted for itself.

```
s/\./c<return>
a c \ & % ^ b
s/\\/d<return>
a c d & % ^ b
s/&/e<return>
a c d e % ^ b
s/%/f<return>
a c d e f ^ b
```

The & and % are only special characters in the replacement text.

```
s/\^/g<return>
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 and then read that file into the buffer containing the other file.

```
ed exer<return>
725
10,17 w temp<return>
210
q<return>
ed exer6<return>
305
7r temp<return>
210
```

The file **temp** can be called any file name.

6-2.

```
33,46m3<return>
```

6-3.

```
.=<return>
10
13p<return>
This is line 13.
j<return>
.p<return>
This is line 13.and line 14.
```

Remember that .= gives you the current line.

# Exercise 7

7-1.

```
% ed newfile1<return>
? newfile1
f current1<return>
current1
a<return>
This is a line of text<return>
Will it go into newfile1<return>
or into current1<return>
.<return>
w<return>
66
q<return>
% ls<return>
bin
current1
```

7-2.

```
ed file1<return>
? file1
a<return>
I am adding text to this file.<return>
Will it show up in ed.hup?<return>
.<return>
```

Turn off your terminal.

Log in again.

```
ed ed.hup<return>
58
f file1<return>
file1
1,$p<return>
I am adding text to this file.
Will it show up in ed.hup?
```

7-3.

```
% ed file1<return>
58
! mail mylogin<return>
You will get mail when<return>
you are done editing!<return>
.<return>
!
```

# The Bourne Shell

This chapter describes how to use the UNIX system 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 section, "Shell Command Language," covers in detail 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 in detail 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 UNIX system and recreate the examples as you read the text. As in the other examples in this guide, different type (**bold,** *italic,* and `typewriter font`) is used to distinguish your input from the UNIX system's output. See "Notation Conventions" in the Preface for details.

In addition to the examples, there are exercises at the end of both the "Shell Command Language" and "Shell Programming" sections. The exercises can help you better understand the topics discussed. The answers to the exercises are at the end of the chapter.


Your UNIX system might 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 C, 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

This chapter covers the characters that have special meanings to the shell and the commands and notations needed to carry out the tasks listed above. Figure 7-1 summarizes the characters with special meanings discussed in this chapter.

| 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 input for a command to come from a file |
| >> | redirects output of a command to be 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. |

Figure 7-1: Characters with Special Meanings in the Shell Language

# 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, and so on.

## The Asterisk (*) Metacharacter

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:

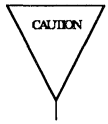
```
echo *<return>
```

The **echo** command displays its arguments on your screen. Notice that the system response to **echo** * is a listing of all the file names in your current directory. However, the file names are displayed horizontally rather than in vertical columns such as those produced by the **ls** command.

Figure 7-2 summarizes the syntax and capabilities of the **echo** command.

| Command Recap<br>**echo** – write any arguments to the output | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **echo** | none | any character(s) |
| Description: | **echo** writes arguments, which are separated by blanks and ended with **<return>** , 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 will be discussed later in this chapter. | |

Figure 7-2: Summary of the **echo** Command




The * is a powerful character. For example, if you type **rm *** you will erase all the files in your current directory. Be very careful how you use it!

For another example, say you have written several reports and have 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*<return>
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*<return>**

Now try another exercise. Choose a character that all the file names in your current directory have in common, such as 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*<return>**

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<return>**

The system response will be 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.

## The Question Mark (?) Metacharacter

The question mark (?) matches any single character of a file name. Let's say 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, as shown below:

```
$ ls chapter?<return>
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??<return>**

Of course, 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<ctrl-g>1<return>**
3. Type: **ls trial1<return>**

   The system will respond with an error message:

   ```
   $ ls trial1<return>
   trial1: no such file or directory
   $
   ```

4. Type: **ls trial?1<return>**

   The system will respond 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<return>
   $ ls trial1<return>
   trial1
   $
   ```

## The Bracket ([ ]) Metacharacters

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<return>
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, if you specify

**chapter[1-5]**

the shell will match any files named **chapter1** through **chapter5**. 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]<return>**

This command will print 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 will look only for uppercase letters; if **[a-z]**, only lowercase letters.

The uses of the metacharacters are summarized in Figure 7-3. Try the metacharacters on the files in your current directory.

| Character | Function |
|---|---|
| * | matches any string of characters, including an empty (null) string |
| ? | matches any single character |
| [ ] | matches one of the sequence of characters specified within the brackets |
| [ - ] | matches one of the range of characters specified |

Figure 7-3: Summary of Metacharacters

# 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."

## The Ampersand (&)

Some shell commands take considerable time to execute. The ampersand (&) is used to execute commands in background mode, thus freeing your terminal for other tasks. The general format for running a command in background mode is

*command* **&<return>**


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 &<return>
21940
$
```

When you run a command in the background, the UNIX 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.)

## The Semicolon (;)

You can type two or more commands on one line as long as each pair is separated by a semicolon (;) , as follows:

*command1*; *command2*; *command3*<**return**>

The UNIX 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<return>**

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 system's responses to these commands to appear on your screen, refer to "Redirecting Output" for instructions.

## The 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<return>
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.

## 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), which 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 line

```
He really wondered why? Why???
```

you could use the **grep** command to match the line with the three question marks as follows:

```
$ grep '???' trial<return>
He really wondered why? Why???
$
```

If you had instead entered the command

```
grep ??? trial<return>
```

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<return>
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 especially 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** will use 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 in double quotes. For example, to send a birthday greeting to another user, type:

**banner happy birthday to you<return>**

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"<return>**

Notice that the words "to" and "you" now appear on the same line. The space between them has lost its meaning as a delimiter.

Figure 7-4 summarizes the syntax and capabilities of the **banner** command.

| Command Recap<br>**banner** – make posters | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **banner** | none | characters |
| **Description:** | **banner** displays up to 10 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. | |

Figure 7-4: Summary of the **banner** Command

## Input and Output Redirection

In the UNIX 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 UNIX 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 rather than the keyboard
- send its output to a file rather than 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: the < Sign

To redirect input, specify a file name after a less than sign (<) on a command line:

*command* < *file*<**return**>

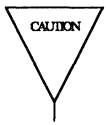
For example, assume that you want to use the **mail**(1) command (described in Chapter 8) to send a message to another user with the login **colleague** and that you already have the message in a file named **report**. You can avoid retyping the message by specifying the file name as the source of input:

**mail colleague < report<return>**

## Redirecting Output to a File: the > Sign

To redirect output, specify a file name after the greater than sign (>) on a command line:

*command* > *file*<**return**>




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 will overwrite the contents of the existing file with the output of your command if you redirect the output to a file with the existing file's 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 will 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<return>
temp
$ ls junk<return>
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 a File: the >> Symbol

To keep from destroying an existing file, you can also use the double redirection symbol (**>>**), as follows:

*command* **>>** *filename***<return>**

This appends the output of a command to the end of the file *filename*. If *filename* 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<return>
This is the first line of trial1.
Hello.
This is the last line of trial1.
$
$ cat trial2<return>
This is the beginning of trial2.
Hello.
This is the end of trial2.
$
$ cat trial2 >> trial1<return>
$ cat trial1<return>
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 (such as a person's name), it will report that as a misspelling, too.

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 will roll 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<return>
```

Figure 7-5 summarizes the syntax and capabilities of the **spell** command.

| Command Recap<br>**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 *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 7-5: Summary of the **spell** Command

### 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

**sort list > list<return>**

the shell will empty **list** and then sort nothing into **list.**

## Background Mode and Output Redirection

Running a command in background does not affect the command's 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 will be 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 &<return>**

You can then use your terminal for other work and examine **testfile** when you have finished it.

## Redirecting Output to a Command: the 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*...**<return>**

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, say 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<return>**

## 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, counting from the left-hand margin.

For example, say 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<return>**

The output is shown below:

```
12/26
7/4
10/18
11/9
4/23
8/12
```

Figure 7-6 summarizes the syntax and capabilities of the **cut** command.

| Command Recap<br>**cut** – cut out selected fields from each line of a file | | |
|---|---|---|
| *command* | options | arguments |
| **cut** | –clist<br>–flist [–**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**<br><br>–**f** lists the field number from the left separated by a delimiter described by –**d**.<br><br>–**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** command and the **split** command. | |

Figure 7-6: Summary of the **cut** Command

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<return>
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<return>
13:14:56
```

Figure 7-7 summarizes the syntax and capabilities of the **date** command.

| **Command Recap**<br>**date** - display the date and time | | |
|---|---|---|
| *command* | options | arguments |
| **date** | **+%m%d%y***<br>**+%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 such as:<br>**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. | |

Figure 7-7: Summary of the **date** Command

* See the **date**(1) manual page in the *IRIS-4D 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`<return>
```

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

### Running Commands with batch and at

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 the terminal; commands entered from the terminal must be ended by pressing **<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<return>**
*first command*<return>
.
.
.
*last command*<return>
**<ctrl-d>**

If there is only one command to be run with **batch**, you can enter it as follows:

**batch** *command_line*<return>
**<ctrl-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<return>
<ctrl-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.

Figure 7-8 summarizes the syntax and capabilities of the **batch** Command.

| Command Recap<br>**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 **<ctrl-d>**. | |

Figure 7-8: Summary of the **batch** Command

The **at** command allows you to specify an exact time to execute the commands. The general format for the **at** command is

**at** *time***<return>**
*first command***<return>**
.
.
.
*last command***<return>**
**<ctrl-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<return>
banner happy birthday | mail emily<return>
<ctrl-d>
job 453400603.a at Thurs Feb 27 08:15:00 1986
$
```

Notice that the **at** command, like the **batch** command, 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***<return>**

Try erasing the previous **at** job for the happy birthday banner. Type in:

**at –r 453400603.a<return>**

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:

```
$ at –l<return>
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<return>
mail mylogin < memo<return>
<ctrl-d>
job 263131754.a at Jun 30 12:00:00 1986
$
$ at –l<return>
user = mylogin 263131754.a at Jun 30 12:00:00 1986
$
```

Figure 7-9 summarizes the syntax and capabilities of the **at** command.

**Command Recap**

**at** – execute commands at a specified time

| *command* | *options* | *arguments* |
|---|---|---|
| **at** | **–r** | *time (date)* |
| | **–l** | *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 *IRIS-4D 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**

Figure 7-9: Summary of the **at** Command

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 UNIX system assigns to each active process.

In the following example, **grep** is run in the background, and 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 &<return>
28223
$
$ ps<return>
PID              TTY   TIME COMMAND
28124            tty10 0:00   sh
28223            tty10 0:04   grep
28224            tty10 0:04   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.

Figure 7-10 summarizes the syntax and capabilities of the **ps** command.

| Command Recap<br>**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 *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 7-10: Summary of the **ps** Command

## Terminating Active Processes

The **kill** command is used to terminate active shell processes. The general format for the **kill** command is

**kill** *PID***<return>**

You can use the **kill** command to terminate processes that are running in background. Note that you cannot terminate background processes by pressing **<break>** or **<delete>**.

The following example shows how you can terminate the **grep** command that you started executing in background in the previous example.

```
$ kill 28223<return>
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
```

Figure 7-11 summarizes the syntax and capabilities of the **kill** command.

| **Command Recap**<br>**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 *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 7-11: Summary of the **kill** Command

## 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* **&<return>**

Notice that you place the **nohup** command before the command you intend to run as a background process.

For example, say you want the **grep** command to search all the 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 & <return>**

You can terminate the **nohup** command by using the **kill** command. Figure 7-12 summarizes the syntax and capabilities 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. | |

Figure 7-12: Summary of the **nohup** Command

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]*<return>**
**echo *<return>**

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<return>**

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.

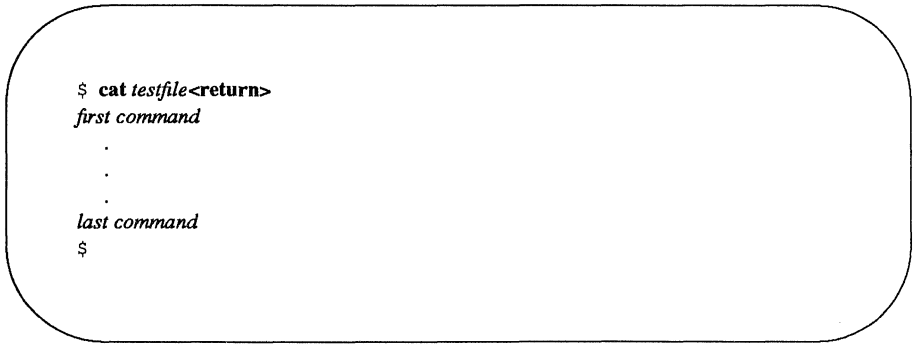
1-7. Instead of cutting the time out of the **date** response, try redirecting only the date, without the time, into **banner**. What is the only part you need to change in the time command line?

**banner 'date | cut –c12-19'<return>**

# Shell Programming

You can use the shell to create programs—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<return>
program_output
$
```

You should be familiar with an editor before you try to create shell programs. Refer to the tutorials in Chapter 5 (for the **vi** editor) and Chapter 6 (for the **ed** editor).

# Shell Programs

## Creating a Simple Shell Program

We will begin by creating a simple shell program that will do the following tasks.

- print the current directory
- list the contents of that directory
- 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 choice of editor, and enter the following:

**pwd<return>**
**ls<return>**
**echo This is the end of the shell program.<return>**

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<return>
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<return>**

The **dl** command is executed by **sh**, and the pathname 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<return>
$ ls –l<return>
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<return>**

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<return>**
**mkdir bin<return>**
**mv dl bin/dl<return>**

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<return>**

What happened?

Figure 7-13 summarizes your new shell program, **dl.**

| **Shell Program Recap**<br>**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.** |

Figure 7-13: Summary of the **dl** Shell Program

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 will execute 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.

One way to keep this from happening is to give the pathname for the system's **ls** command, **/bin/ls**, when you write your own shell program.

The following **ls** shell program would work:

```
$ cat ls<return>
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 pathname, **/bin/ls**.

# Variables

Shell programs manipulate variables as well as 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 **$**: **$1**, **$2**, **$3**, and so on.

A shell program may reference up to nine positional parameters. If a shell program is invoked on a command line that appears like this:

**shell.prog pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9<return>**

then positional parameter **$1** within the program will be assigned the value **pp1**, positional parameter **$2** within the program will be assigned the value **pp2**, and so on.

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<return>
echo  The first positional parameter is: $1<return>
echo  The second positional parameter is: $2<return>
echo  The third positional parameter is: $3<return>
echo  The fourth positional parameter is: $4<return>
$
```

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<return>
$
$ pp one two three four<return>
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<return>
banner happy birthday | mail $1
```

Try sending yourself a birthday greeting. If your login name is **sue**, your command line will be:

**bbday sue<return>**

Figure 7-14 summarizes the syntax and capabilities of the **bbday** shell program.

| Shell Program Recap<br>**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. |

Figure 7-14: Summary of the **bbday** Command

The **who** command lists all users currently logged in to 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<return>**

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 <return>**

The output is shown on the following screen:

```
$ whoson sue<return>
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.

Figure 7-15 summarizes the syntax and capabilities of the **whoson** command.

| **Shell Program Recap**<br>**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. |

Figure 7-15: Summary of the **whoson** Command

The shell allows a command line to contains 128 arguments. However, a shell program is restricted to referencing nine positional parameters, **$1** through **$9**, at a given time. The special parameter **$***, described in the next section, can also be used to access the values of all command line arguments.

## Special Parameters

**$#** This 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<return>
echo The number of arguments is: $#
$
```

The program simply displays the number of arguments with which it is invoked. For example:

```
$ get.num test out this program<return>
The number of arguments is: 4
$
```

Figure 7-16 summarizes the **get.num** shell program.

| Shell Program Recap<br>**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 **$#**. |

Figure 7-16: Summary of the **get.num** Shell Program

**$*** This 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 the parameters. Use the command line shown in the following completed file:

```
$ cat show.param<return>
echo The parameters for this command are: $*
$
```

**show.param** will echo 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?<return>
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<return>
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, say 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?<return>
The parameters for this command are: chap1 chap2 chap3
chap4 chap5 chap6 chap7
$
```

Figure 7-17 summarizes the **show.param** shell program.

| **Shell Program Recap**<br>**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. |

Figure 7-17: Summary of the **show.param** Shell Program

## Named Variables

Another form of variable that 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*<**return**>

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<return>**

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.
- **HOME** is the default variable for the **cd** command (home directory).
- **IFS** defines the internal field separators (normally the space, the tab, and the 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 if you are editing with **vi**.
- **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 *IRIS-4D 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.

## 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***<return>**

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 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***<return>**

The values assigned by **read** to *variable* will be 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 such as `Type in ....` The **read** command will wait until you type a character string, followed by **<return>**, and then make 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 person's 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 one displayed here:

```
$ cat num.please<return>
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 person's name
- **read** assigns the person's name to the variable *name*
- **echo** asks for the person's 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 will be ready to make it executable (with the **chmod** command):

```
$ cat mknum<return>
echo Type in name
read name
echo Type in number
read num
echo $name $num >> list
$ chmod u+x mknum<return>
$
```

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<return>
Type in the name
Mr. Niceguy<return>
Type in the number
668-0007<return>
$ num.please<return>
Type in last name
Niceguy<return>
Mr. Niceguy 668-0007
$
```

Notice that the variable **name** accepts both **Mr.** and **Niceguy** as the value.

Figures 7-18 and 7-19 summarize the **mknum** and **num.please** shell programs, respectively.

| Shell Program Recap<br>**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. |

Figure 7-18: Summary of the **mknum** Shell Program

| Shell Program Recap<br>**num.please** – display a person's name and number | |
|---|---|
| *command* | *arguments* |
| **num.please** | (interactive) |
| **Description:** | Asks you for a person's last name, and then displays the person's full name and telephone number. |
| **Remarks:** | This is an interactive command. |

Figure 7-19: Summary of the **num.please** Shell Program

### Substituting Command Output for the Value of a Variable

You can substitute a command's output for the value of a variable by using *command substitution*. This has the following format:

*variable*=`*command*`**<return>**

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 the following:

**date | cut –c12-19<return>**

You can put this in a simple shell program called **t** that will give you the time.

```
$ cat t<return>
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<return>
$ t<return>
The time is: 10:36
$
```

Figure 7-20 summarizes your **t** program.

| Shell Program Recap<br>**t** – display the correct time | |
|---|---|
| *command* | *arguments* |
| **t** | none |
| **Description:** | **t** gives you the correct time in hours and minutes. |

Figure 7-20: Summary of the **t** Shell Program

### Assigning Values with Positional Parameters

You can assign a positional parameter to a named parameter by using the following format:

*var1* =**$1<return>**

The next example is a simple program called **simp.p** that assigns a positional parameter to a variable. The following screen shows the commands in **simp.p**:

```
$ cat simp.p<return>
var1=$1
echo $var1
$
```

Of course, you can also assign the output of a command that uses positional parameters to a variable, as follows:

**person=`who | grep $1`**<return>

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<return>
person=`who | grep $`
echo $person >> login.file
echo $person
$
```

The system response to **log.time** is shown in the following screen:

```
$ log.time maryann<return>
maryann     tty61          Apr 11 10:26
$
```

Figure 7-21 summarizes the **log.time** shell program.

| Shell Program Recap<br>**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. |

Figure 7-21: Summary of the **log.time** Shell Program

## Shell Programming Constructs

The shell programming language has several constructs that give added flexibility to your programs:

- Comments let you document a program's 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*<**return**>

For example, a program that contains the following lines will ignore them when it is executed:

```
# This program sends a generic birthday greeting.<return>
# This program needs a login as<return>
# the positional parameter.<return>
```

Comments are useful for documenting a program's function and should be included in any program you write.

## 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 7-22 shows the general format for a here document.

center,box; l.

*command* <<*delimiter*<**return**> *...input lines...*<**return**> *delimiter*<**return**>

Figure 7-22: Format of a here Document

```
# This program sends a generic birthday greeting.<return>
# This program needs a login as<return>
# the positional parameter.<return>
```

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<return>
mail $ <<!
Best wishes to you on your birthday.
!
$
```

When you use this command, you must specify the recipient's 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<return>
```

Login **mary** will receive your greeting the next time she reads her mail messages:

```
$ mail<return>
From mylogin Wed May 14 14:31 CDT 1986
Best wishes to you on your birthday
$
```

Figure 7-23 summarizes the format and capabilities of the **gbday** command.

| **Shell Program Recap**<br>**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. |

Figure 7-23: Summary of the **gbday** Command

## 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** which does these tasks.

```
$ cat ch.text<return>
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<return>**

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<return>
Type in the filename.
memo<return>
Type in the exact text to be changed.
Dear John:<return>
Type in the exact new text to replace the above.
To whom it may concern:<return>
$ cat memo<return>
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.

Figure 7-24 summarizes the format and capabilities of the **ch.text** command.

| Shell Program Recap<br>**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 will prompt you to type in the arguments. |

Figure 7-24: Summary of the **ch.text** Command

If you want to become more familiar with **ed**, see Chapter 6, "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) than 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. For more information about **exit**, 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<return>
        in a_list_of_values<return>
do<return>
        command 1<return>
        command 2<return>
                .
                .
                .
        last command<return>
done<return>
```

Figure 7-25: 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 will execute the next line below **done**. If there is no line, the program will end.

The easiest way to understand a shell programming construct is to try an example. Create a program that will move files to another directory. Include the following commands for the purposes shown:

| | |
|---|---|
| **echo** | to prompt the user for a pathname to the new directory. |
| **read** | to assign the pathname 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 the arguments entered on the command line). |
| **do** *command_sequence* | to provide a command sequence. The construct for this program will be: |

**do**
**mv** **$***file* **$***path*/**$***file*<return>
**done**

The following screen shows the text for the shell program **mv.file**:

```
$ cat mv.file<return>
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<return>
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).

Figure 7-26 summarizes the **mv.file** shell program.

| **Shell Program Recap**<br>**mv.file** – move files to another directory (user defined) | |
|---|---|
| *command* | *arguments* |
| **mv.file** | *filenames*<br>(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. |

Figure 7-26: Summary of **mv.file** Shell Program

### 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 7-27.

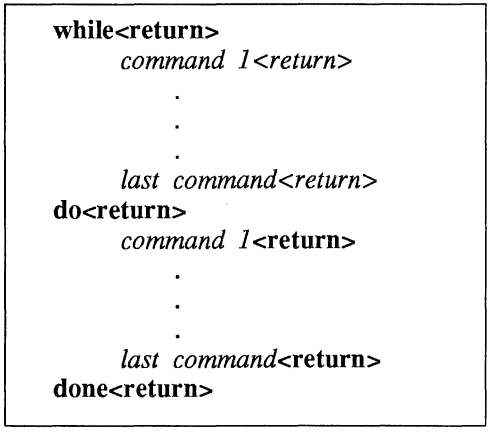



Figure 7-27: Format of the **while** Loop Construct

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<return>
while
   read x
do
   echo $x>>xfile
done
$
```

With some added refinements, the program becomes:

```
$ cat enter.name<return>
echo Please type in each person's name and then a <return>
echo Please end the list of names with a <ctrl-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<return>
Please type in each person's name and then a <return>
Please end the list of names with a <ctrl-d>
Mary Lou<return>
Janice<return>
<ctrl-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<return>
```

Notice that the system responded with a prompt. The output from the **who** command was placed in **/dev/null** and was effectively discarded.

## Conditional Constructs

### 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 7-28.

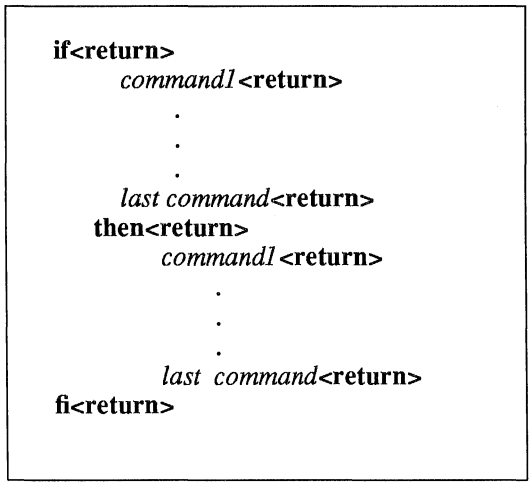



Figure 7-28: Format of the **if...then** Conditional Construct

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<return>
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<return>**

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:

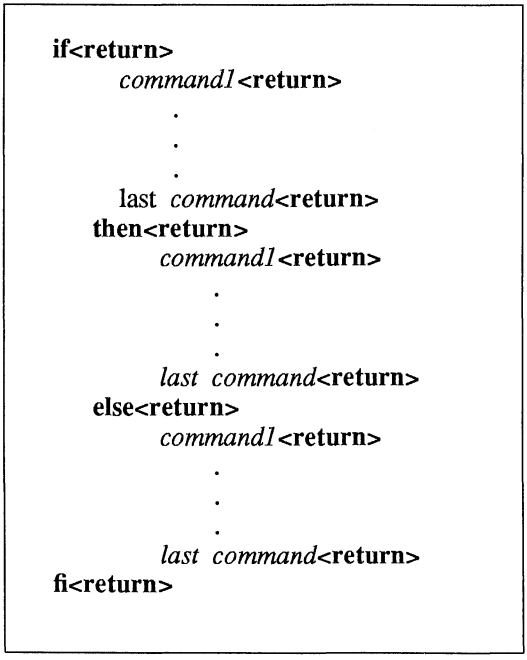



Figure 7-29: Format of the **if...then...else** Conditional Construct

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<return>
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
$
```

Figure 7-30 summarizes your enhanced **search** program.

| **Shell Program Recap**<br>**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) |

Figure 7-30: Summary of the **search** Shell Program

### 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 will continue. If the condition is false, the loop will end and the next command will be executed. Some of the useful options for the **test** command are:

| | |
|---|---|
| **test –r** *file***<return>** | true if the file exists and is readable |
| **test –w** *file***<return>** | true if the file exists and has write permission |
| **test –x** *file***<return>** | true if the file exists and is executable |
| **test –s** *file***<return>** | true if the file exists and has at least one character |
| **test** *var1* **–eq** *var2***<return>** | true if *var1* equals *var2* |
| **test** *var1* **–ne** *var2***<return>** | 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<return>
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. Your program will be as follows:

```
$ cat mv.ex<return>
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 will 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<return>
for file
  do
    if test -x $file
       then
         mv $file $HOME/bin/$file
    fi
  done
$
```

Test the command, using all the files in the current directory, specified with the * metacharacter as the command argument. The command lines shown in the following example executes the command from the current directory and then changes to **bin** and lists the files in that directory. All executable files should be there.

```
$ mv.ex *<return>
$ cd; cd bin; ls<return>
list_of_executable_files
$
```

Figure 7-31 summarizes the format and capabilities of the **mv.ex** shell program.

| **Shell Program Recap**<br>**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. |

Figure 7-31: Summary of the **mv.ex** Shell Program

**case..esac**

The **case...esac** construction has a multiple choice format that allows you to choose one of several patterns and then execute a list of commands for that pattern. The pattern statements must begin with the keyword **in**, and a ) must be placed after the last character of each pattern. The command sequence for each pattern is ended with **;;**. The **case** construction must be ended with **esac** (the letters of the word case reversed).

The general format for the **case** construction shown in Figure 7-32:

```
case word<return>
in<return>
    pattern1)<return>
        command line 1<return>
            .
            .
            .
        last command line<return>
    ;;<return>
    pattern2)<return>
        command line 1<return>
            .
            .
            .
        last command line<return>
    ;;<return>
    pattern3)<return>
        command line 1<return>
            .
            .
            .
        last command line<return>
    ;;<return>
    *)<return>
        command 1<return>
            .
            .
            .
        last command<return>
    ;;<return>
esac<return>
```

Figure 7-32: 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 so 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 shell's 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***<return>**

(For an explanation of the commands used, see the **vi** tutorial in Chapter 5.) 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 will warn that you do not have a pattern for the terminal specified and will allow you to exit the **case** construct:

```
$ cat set.term<return>
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.

Figure 7-33 summarizes the format and capabilities of the **set.term** shell program.

| Shell Program Recap<br>**set.term** - assign a value to TERM (user defined) | |
|---|---|
| *command* | *arguments* |
| **set.term** | interactive |
| **Description:** | Assigns a value to the shell variable TERM and 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. |

Figure 7-33: Summary of the **set.term** Shell Program

## Unconditional Control Statements

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 the next example shows:

```
$ cat set.term<return>
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<return>
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<return>
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 delete key to end the program.

To debug this program, try executing **bug** using **sh –v**. This will print the lines of the file as they are read by the system, as shown below:

```
$ sh –v bug tom<return>
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<return>
+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 as follows:

```
$ cat bug<return>
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**

**saverfile** is the file that saves the output of *command1* for you to study.

For example, say you want to check on the output of the **grep** command in the following command line:

**who | grep $1 | cut –c1-9<return>**

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<return>**

The file **check** contains a copy of the **grep** output, as shown in the following screen:

```
$ who | grep mlhmo | tee check | cut –c1-9<return>
mlhmo
$ cat check<return>
mlhmo    tty61    Apr 10    11:30
$
```

# Modifying Your Login Environment

The UNIX system lets you modify your login environment in several ways. One modification that users commonly want to make is to change the default values of the erase (#) and line kill (@) characters.

When you log in, the shell first examines a file in your login directory named **.profile** (pronounced "dot 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<return>**

You can add commands to your **.profile** just as you add commands to any other shell program.

## 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 will reinitialize your environment by reading executing the commands in your **.profile**. Try this now. Type:

**. .profile<return>**

The system should respond with the following:

```
Good Morning! I am ready to work for you.
$
```

## Setting Terminal Options

The **stty** command can make your shell environment more convenient. There are three options you can use with **stty**: **–tabs**, **erase <ctrl-h>**, and **echoe**.

| | |
|---|---|
| **stty –tabs** | This option preserves tabs when you are printing. It 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 *IRIS-4D User's Reference Manual* for details.) |
| **stty erase <ctrl-h>** | This option allows you to use the erase key on your keyboard to erase a letter, instead of the default character #. Usually the backspace key is the erase key. |
| **stty echoe** | If you have a terminal with a screen, this option erases characters from the screen as you erase them with the backspace key. |

If you want to use these options for the **stty** command, you can create those 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 those four command lines to your **.profile**:

```
$ tail –4 .profile<return>
echo Good Morning! I am ready to work for you
stty –tabs
stty erase <ctrl-h>
stty echoe
$
```

Figure 7-34 summarizes the format and capabilities of the **tail** command.

| Command Recap<br>**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 ten lines). You can specify a number of blocks (*–n*b) or characters (*–n*c) instead of lines. | |

Figure 7-34: Summary of the **tail** Command

## 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 pathname of your login directory. Use the **cd** command to go to your login directory and type:

**pwd<return>**

What was the system response? Now type:

**echo $HOME<return>**

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<return>**

The system will respond with your current PATH value.

```
$ echo $PATH<return>
:/mylogin/bin:/bin:/usr/bin:/usr/lib
$
```

The colon ( **:** ) is a delimiter between pathnames in the string assigned to the $PATH variable. When nothing is specified before a **:** , then 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"<return>**

### 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*<return>**
**export TERM<return>**
**tput init**

The first two lines, together, are necessary to tell the computer what type of terminal you are using. The last line, containing the **tput** 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 will be executed automatically whenever you log in. To determine what terminal name to assign to the TERM variable, see the instructions in Appendix D, "Setting Up the Terminal." This appendix also contains details about the **tput** command.

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<return> "**

Now execute your **.profile** (with the . command) and watch for your new prompt sign.

```
$ . .profile<return>
Your command is my wish
```

The mundane $ sign is gone forever, or at least until you delete the **PS1** variable from your **.profile**.

# Shell Programming Exercises

2-1. Create a shell program called **time** from the following command line:

**banner `date | cut –c12-19`<return>**

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 UNIX 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. Set the **stty** options for your environment.

2-11. Change your prompt to the word Hello.

2-12. Check the settings of the variables **$HOME**, **$TERM**, and **$PATH** in your environment.

# Answers To Exercises

## 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<return>
cat
123t
new.t
t
$
```

1-2. The command **cat [0-9]*** will produce the following output:

```
1memo
100data
9
05name
```

The command **echo *** will produce a list of all the files in the current directory.

1-3. You can place **?** in any position in a file name.

1-4. The command **ls [0-9]*** will list only those files that start with a number.

The command **ls [a-m]*** will list 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<return>**

1-7. Change the **–c** option of the command line to read:

**banner `date | cut –c1-10`<return>**

# Shell Programming Exercises

2-1.

```
$ cat time<return>
banner `date | cut –c12-19`
$
$ chmod u+x time<return>
$ time<return>
(banner display of the time 10:26)
$
```

2-2.

```
$ cat mydate<return>
banner `date | cut -c1-10`
$
```

2-3.

```
$ cat tofriends<return>
echo Type in the name of the file containing the note.
read note
mail janice marylou bryan < $note
$
```

Or, if you used parameters for the logins, instead of the logins themselves, your program may have looked like this:

```
$ cat tofriends<return>
echo Type in the name of the file containing the note.
read note
mail $* < $note
$
```

2-4. **date | cut –c1-10 > file1<return>**

2-5. **echo Dear colleague >> file1<return>**

2-6.

```
$ cat send.memo<return>
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<return>
echo type in the directory path
read path
echo type in file names, end with <ctrl-d>
  while
  read file
    do
      mv $file $path/$file
    done
echo all done
$
```

2-8.

```
$ cat mv.file<return>
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 *<return>**

2-9. See hint given with exercise 2-9.

```
$ cat search<return>
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 lines to your **.profile.**

**stty –tabs<return>**
**stty erase <ctrl-h><return>**
**stty echoe<return>**

2-11. Add the following command lines to your **.profile**

**PS1=Hello<return>**
**export PS1**

2-12. To check the values of these variables in your home environment:

- $ **echo $HOME<return>**
- $ **echo $TERM<return>**
- $ **echo $PATH<return>**

# The C Shell

**csh** is a new command language interpreter for UNIX systems. It incorporates good features of other shells and a history mechanism similar to the redo of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to **csh** are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with **csh** is possible after reading just the first section of this chapter. The second section describes the shell's capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

This chapter includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

# The C Shell Language Interpreter

A shell is a command language interpreter. **csh** is the name of one particular command interpreter on UNIX. The primary purpose of **csh** is to translate command lines typed at a terminal into system actions, such as invocation of other programs. **csh** is a user program just like any you might write. Hopefully, **csh** will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the *IRIS-4D Programmer's Manual*. The **csh** documentation in the manual provides a full description of all features of the shell and is a final reference for questions about the shell.

## Terminal Usage of the Shell

### The Basic Notion of Commands

A shell in UNIX acts mostly as a medium through which other programs are invoked. While it has a set of built-in functions which it performs directly, most commands cause execution of programs that are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system consist of a list of strings or words interpreted as a command name followed by arguments. Thus the command

**mail bill**

consists of two words. The first word, *mail*, names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to execute it for you. It will look in a number of directories for a file with the name *mail* which is expected to contain the *mail* program.

The rest of the words of the command are given as arguments to the command itself when it is executed. In this case we specified also the argument **bill** which is interpreted by the mail program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the **mail** command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
        Bill<ctrl-D>
%
```

Here we typed a message to send to **bill** and ended this message with a **ctrl-D**. The mail program then echoed the characters 'EOT' and transmitted our message. The characters '%' were printed before and after the **mail** command by the shell to indicate that input was needed.

After typing the '%' prompt the shell was reading command input from our terminal. We typed a complete command

**mail bill**

The shell then executed the mail program with argument **bill** and went dormant waiting for it to complete. The mail program then read input from our terminal until we signaled an end-of-file by typing a **ctrl-D**. The shell noticed that mail had completed and signaled us that it was ready to read from the terminal again by printing another '%' prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes, it prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

An example of a useful command you can execute now is the **tset** command, which sets the default **erase** and **kill** characters on your terminal – the erase character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, the erase character is '#' and the kill character is '@'. Most people who use CRT displays prefer to use the backspace character as their erase character since it is then easier to see what you have typed so far. To do this type:

**tset**

This tells the program **tset** to set the erase character, and its default setting for this character is a backspace.

## Optional Arguments

A useful notion in UNIX is that of a flag argument. While many arguments to commands specify file names or user names, some arguments specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character '–' (hyphen). Thus the command

**ls**

will produce a list of the files in the current working directory. The option **–s** is the size option, and the command

**ls –s**

causes **ls** to also give, for each file the size of the file in blocks of 512 characters. The manual section for each command in the *UNIX Reference Manual* gives the available options for each command. The **ls** command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard-to-remember options.

## Redirecting Output to Files

Commands that normally read input or write output on the terminal can also be executed with this input and/or output done to a file.

Suppose we wish to save the current date in a file called **now**. The command

**date**

will print the current date on our terminal. This is because our terminal is the default standard output for the date command. The shell lets us redirect the standard output of a command through a notation using the metacharacter '>' and the name of the file where output is to be placed.

Thus, the command

**date** > *now*

This command places the current date and time into the file *now*. It is important to know that the date command was unaware that its output was going to a file rather than to the terminal. The shell performed this redirection before the command began executing.

One other thing to note here is that the file *now* need not have existed before the date command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! The shell option **noclobber** exists to prevent this from happening accidentally.

The system normally keeps files which you create with '>' and all other files. Thus the default is for files to be permanent. If you wish to create a file which will be removed automatically, you can begin its name with a '#' character, this 'scratch' character denotes the fact that the file will be a scratch file. The system will remove such files after a couple of days, or sooner if file space becomes very tight. Thus, in running the date command above, we don't really want to save the output forever, so we would more likely do

**date > *#now***

## Special Characters in the Shell

The shell has a large number of special characters (like '>') which indicate special functions. We say that these notations have syntactic and semantic meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of quotation which allows us to use metacharacters without the shell treating them in any special way.

Metacharacters normally have effect only when the shell is reading our input. We need not worry about placing shell metacharacters in a letter we are sending via mail, or when we are typing in text or data to some other program. Note that the shell is only reading input when it has prompted with '% '.

## Redirecting Input From a File

We learned above how to redirect the standard output of a command to a file. It is also possible to redirect the standard input of a command from a file. This is not often necessary since most commands will read from a file whose name is given as an argument. We can give the command

**sort < data**

to run the **sort** command with standard input, where the command normally reads its input, from the file *data*. We would more likely type

**sort data**

letting the **sort** command open the file *data* for input itself since this is less to type.

We should note that if we just typed **sort** then the sort program would sort lines from its standard input. Since we did not redirect the standard input, it would sort lines as we typed them on the terminal until we typed a **ctrl-D** to indicate an end-of-file.

Combining the standard output of one command with the standard input of another or running the commands in a sequence known as a pipeline. For instance the command

**ls –s**

normally produces a list of the files in our directory with the size of each in blocks of 512 characters. Combining the sort command with the **ls** argument allows the user to sort differently .

The –n option of sort specifies a numeric sort rather than an alphabetic sort. Thus

**ls –s | sort –n**

specifies that the output of the **ls** command run with the option –s is to be piped to the command **sort** run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the reverse sort option **–r** and the head command in combination with the previous command.

**ls –s | sort –n –r | head –5**

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the sort command asking it to sort numerically in reverse order (largest first). This output has then been run into the command **head** which gives us the first few lines. In this case we have asked**head** for the first five lines. This command gives us the names and sizes of our five largest files.

The notation introduced above is called the pipe mechanism. Commands separated by '|' characters are connected together by the shell and the standard output of each is run into the standard input of the next.

The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes is in the routing of information to the line printer.

## Filenames

Many commands to be executed will need the names of files as arguments. UNIX pathnames consist of a number of components separated by '/'. Each component except the last names a directory in which the next component resides, in effect specifying the path of directories to follow to reach the file. Thus the pathname

**/etc/motd**

specifies a file in the directory **etc** which is a subdirectory of the root directory '/'. Within this directory the file named is *motd* which stands for 'message of the day'.

A pathname that begins with a slash is said to be an absolute pathname since it is specified from the absolute top of the entire directory hierarchy of the system (the root ). Pathnames which do not begin with '/' are interpreted as starting in the current "working directory" , which is, by default, your home directory and can be changed by **cd**, the change directory command.

Such pathnames are said to be "relative" to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each component of the pathname. If the pathname contains no slashes at all then the file is contained in the working directory itself and the pathname is merely the name of the file in this directory. Absolute pathnames have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and '.'s (dots). In fact, all printing characters except '/' (slash) may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character '.' (dot) is not a shell-metacharacter and is often used to separate the extension of a file name from the base of the name. Thus

*prog.c prog.o prog.errs prog.output*

are four related files. They share a base portion of a name (a base portion being that part of the name that is left when a trailing '.' and following characters which are not '.' are stripped off). The file *prog.c* might be the source for a C program, the file *prog.o* the corresponding object file, the file *prog.errs* the errors resulting from a compilation of the program, and the file *prog.output* the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the notation

*prog.**

This word is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with *prog*. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are alphabetically sorted and placed in the argument list of the command. Thus the command

**echo** *prog.**

will echo the names

*prog.c prog.errs prog.o prog.output*

Note that the names are in sorted order here, and a different order than we listed

them above. The echo command receives four words as arguments, even though we only typed one word as argument directly. The four words were generated by filename expansion of the one input word.

Other notations for filename expansion are also available. The character '?' matches any single character in a filename. Thus

**echo ? ?? ???**

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently sorted.

Another mechanism consists of a sequence of characters between '[' and ']'. This metasequence matches any single character from the enclosed set. Thus

*prog.[co]*

will match

*prog.c prog.o*

in the example above. We can also place two characters around a '–' in this notation to denote a range. Thus

*chap.[1–5]*

might match files

*chap.1 chap.2 chap.3 chap.4 chap.5*

if they existed. This is shorthand for

*chap.[12345]*

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an argument list) contains file-name expansion syntax, and if this file-name expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

**No match**

and does not execute the command.

Another very important point is that files with the character '.' at the beginning are treated specially. Neither '*' or '?' or the '[' ']' mechanism will match it. This prevents accidental matching of the filenames '.' and '..' in the working directory which have special meaning to the system, as well as other files such as **.cshrc** which are not normally visible. We will discuss the special role of the file **.cshrc** later.

Another file-name expansion mechanism gives access to the pathname of the home directory of other users. This notation consists of the character ~ (tilde) followed by another users' login name. For instance the word **~bill** would map to the pathname **/usr/bill** if the home directory for *bill* was **/usr/bill**. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a reliable way of accessing the files of other users.

A special case of this notation consists of a ~ alone, e.g. **~/mbox**. This notation is expanded by the shell into the file *mbox* in your home directory, i.e. into **/usr/bill/mbox** for me on Ernie Co-vax, the UCB Computer Science Department VAX machine, where this document was prepared. This can be very useful if you have used **cd** to change to another directory and have found a file you wish to copy using **cp**. If I give the command

**cp** *thatfile* ~

the shell will expand this command to

**cp** *thatfile* **/usr/bill**

since my home directory is **/usr/bill.**

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of words which have common parts but cannot be abbreviated by the above mechanisms because they are the names of files which do not yet exist. This mechanism will be described later, in "Braces in Argument Expansion".

## Arguments Enclosed in Quotations

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus the command

**echo ***

will not echo the character '*'. It will either echo a sorted list of file names in the current working directory, or print the message 'No match' if there are no files in the working directory.

The recommended mechanism for placing characters which are neither numbers, digits, '/', '.', or '–' in an argument word to a command is to enclose it with single quotation characters ''.

**echo '*'**

There is one special character '!' which is used by the history mechanism of the shell and which cannot be escaped by placing it within '' characters. It and the

character '´' itself can be preceded by a single '\' to prevent their special meaning. Thus

**echo \´\!**

prints

´!

These two mechanisms suffice to place any printing character into a word which is an argument to a shell command. They can be combined, as in

**echo \´´*´**

which prints

´*

since the first '\' escaped the first '´' and the '*' was enclosed between '´' characters.

## Terminating Commands

When you are executing a command and the shell is waiting for it to complete there are several ways to force it to stop. For instance if you type the command

**cat /etc/passwd**

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an interrupt signal to the cat command by typing the delete or rubout key on your terminal.

NOTE

Many users use **stty** (1) to change the interrupt character to ↑C.

Since cat does not take any precautions to avoid or otherwise handle this signal, the interrupt will cause it to terminate. The shell notices that cat has terminated and prompts you again with %. If you hit interrupt again, the shell will just repeat its prompt.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the mail program in the first example above was terminated when we typed a **ctrl-D** which generates an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing 'logout'; UNIX then logs you off the system. Since this means that typing **ctrl-D** too many times can accidentally log you off, the shell has a mechanism for preventing this. The **ignoreeof** option will be discussed in "Shell Variables" later in this chapter.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

**mail bill** < *prepared.text*

the mail command will terminate without our typing **ctrl-D**. This is because it read to the end-of-file of our file *prepared.text* in which we placed a message for 'bill' with an editor program. We could also have done

**cat** *prepared.text* | **mail bill**

since the **cat** command would then have written the text through the pipe to the standard input of the mail command. When the **cat** command completed it would have terminated, closing down the pipeline. The **mail** command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an interrupt.

Another possibility for stopping a command is to suspend its execution temporarily, with the possibility of continuing execution later. This is done by sending a stop signal via typing a **ctrl-Z**. This signal causes all commands running on the terminal (usually one but more if a pipeline is executing) to be suspended. The shell notices that the command(s) have been suspended, types 'Stopped' and then prompts for a new command.

The previously executing command has been suspended, but otherwise unaffected by the stop signal. Any other commands can be executed while the original command remains suspended. The suspended command can be continued using the **fg** command with no arguments.

The shell will then retype the command to remind you which command is being continued, and cause the command to resume execution. Unless any input files in use by the suspended command have been changed in the meantime, the suspension has no effect whatsoever on the execution of the command. This feature can be very useful during editing, when you need to look at another file before continuing. An example of command suspension follows.

```
% mail harold
Someone just copied a big file into my directory and its name is
<ctrl-Z>
Stopped
% ls
funnyfile
prog.c
prog.o
% jobs
[1] + Stopped                    mail harold
% fg
mail harold
funnyfile. Do you know who did it?
EOT
%
```

In this example the user was sending a message to Harold and forgot the name of the file he wanted to mention. The mail command was suspended by typing **ctrl-Z**. When the shell noticed that the mail program was suspended, it typed 'Stopped' and prompted for a new command. Then the **ls** command was typed to find out the name of the file. The **jobs** command was run to find out which command was suspended. At this time the **fg** command was typed to continue execution of the mail program. Input to the mail program was then continued and ended with a **ctrl-D** which indicated the end of the message at which time the mail program typed EOT. The **jobs** command will show which commands are suspended. The **ctrl-Z** should only be typed at the beginning of a line since everything typed on the current line is discarded when a signal is sent from the keyboard. This also happens on interrupt and quit signals.

More information on suspending jobs and controlling them is given "Background, Foreground, or Suspending Jobs" later in this chapter.

If you write or run programs which are not fully debugged then it may be necessary to stop them somewhat ungracefully. This can be done by sending them a quit signal, sent by typing a **ctrl-e**. This will usually provoke the shell to produce a message like:

**Quit** (Core dumped)

indicating that a file 'core' has been created containing information about the program's state when it was terminated. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the core

file is.

If you run background commands then these commands will ignore interrupt and quit signals at the terminal. To stop them you must use the **kill** command.

If you want to examine the output of a command without having it move off the screen as the output of the

**cat /etc/passwd**

command will, you can use the command

**more /etc/passwd**

The more program pauses after each complete screenful and types '—More—' at which point you can press the space bar to get another screenful, a return to get another line, or a 'q' to end the more program. You can also use more as a filter, i.e.

**cat /etc/passwd | more**

For stopping output of commands not involving more you can use the **ctrl-S** key to stop the output. The output will resume when you hit **ctrl-Q** or any other key, but ctrl-Q is normally used because it only restarts the output and does not become input to the program which is running. This works well on low-speed terminals, but at 9600 baud it is hard to type **ctrl-S** and **ctrl-Q** fast enough to paginate the output nicely, and a program like more is usually used.

An additional possibility is to use the **ctrl-O** flush output character; when this character is typed, all output from the current command is thrown away until the next input read occurs or until the next shell prompt. This can be used to allow a command to complete without having to suffer through the output on a slow terminal. **ctrl-O** is a toggle, so flushing can be turned off by typing **ctrl-O** again while output is being flushed.

## Working In the C Shell

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to UNIX and type the following command to the system:

**chsh** *myname* **/bin/csh**

Here *myname* should be replaced by the name you typed to the system prompt of 'login:' to get onto the system. You only have to do this once; it takes effect at next login. You are now ready to use **csh**.

Before you do the **chsh** command, the shell you are using when you log into the system is **/bin/sh.** In fact, much of the above discussion is applicable to **/bin/sh.** The next section will introduce many features particular to **csh** so you should change your shell to **csh** before you begin reading it.

# Details on the Shell for Terminal Users

## Shell Startup and Termination

When you login, the shell is started by the system in your home directory and begins by reading commands from a file **.cshrc** in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A login shell, executed after you log in to the system, will, after it reads commands from **.cshrc**, read commands from a file **.login** also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My .login file looks something like:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users" ; users
alias ts \
ts; stty intr ^C kill ^U crt
set time=15 history=10
msgs -f
if (-e $mail) then
                                echo "${prompt}mail"
                                mail
endif
```

This file contains several commands to be executed by UNIX each time I login. The first is a set command which is interpreted directly by the shell. It sets the shell variable **ignoreeof** which causes the shell to not log me off if I hit **ctrl-D**. Rather, I use the logout command to log off of the system. By setting the mail variable, I ask the shell to watch for incoming mail to me. Every 5 minutes the shell looks for this file and tells me if more mail has arrived there. An alternative to this is to put the command

**biff y**

in place of this set; this will cause me to be notified immediately when mail arrives, and to be shown the first few lines of the new message.

Next I set the shell variable 'time' to '15' causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of cpu time. The variable 'history' is set to 10 indicating that I want the shell to remember the last 10 commands I type in its history list (described in "The History List" later in this chapter).

I create an alias ''ts'' which executes a **tset** (1) command setting up the modes of the terminal. The parameters to **tset** indicate the kinds of terminal which I usually use when not on a hardwired port. I then execute ''ts'' and also use the **stty** command to change the interrupt character to **ctrl-C** and the line kill character to **ctrl-U**.

I then run the **msgs** program, which provides me with any system messages which I have not seen before; the '–f' option here prevents it from telling me anything if there are no new messages. Finally, if my mailbox file exists, then I run the **mail** program to process my mail.

When the mail and msgs programs finish, the shell will finish processing my **.login** and begin reading commands from the terminal, prompting for each with %.

When I log off (by giving the logout command) the shell will print 'logout' and execute commands from the file '.logout' if it exists in my home directory. After that, the shell will terminate and UNIX will log me off the system. If the system is not going down, I will receive a new login message. In any case, after the 'logout' message the shell is committed to terminating and will take no further input from my terminal.

## Shell Variables

The shell maintains a set of variables. We saw above the variables **history** and **time** which had values '10' and '15'. In fact, each shell variable has as value an array of zero or more strings. Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

**set name=value**

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable "path". This variable contains a sequence of directory names where the shell searches for commands. The **set** command with no arguments shows the value of all variables currently defined (we usually say set) in the shell. The default value for "path" will be shown by set to be

```
% set
argv        ()
cwd         /usr/bill
home        /usr/bill
path        (. /usr/ucb /bin /usr/bin)
prompt      %
shell       /bin/csh
status      0
term        c100rv4pna
user        bill
%
```

This output indicates that the variable path points to the current directory '.' and then '/usr/ucb', '/bin' and '/usr/bin'. Commands which you may write might be in '.' (usually one of your directories). Commands developed at Berkeley, live in '/usr/ucb' while commands developed at Bell Laboratories live in '/bin' and '/usr/bin'.

A number of locally developed programs on the system live in the directory '/usr/local'. If we wish that all shells which we invoke to have access to these new programs we can place the command

**set path=(. /usr/ucb /bin /usr/bin /usr/local)**

in our file **.cshrc** in our home directory. Try doing this and then logging out and back in and do

**set**

again to see that the value assigned to path has changed.

One thing you should be aware of is that the shell examines each directory which you insert into your path and determines which commands are contained there. Except for the current directory '.', which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found by the shell. If you wish to use a

command which has been added in this way, you should give the command

**rehash**

to the shell, which will cause it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory '.' on each command, placing it at the end of the path specification usually works the same way and reduces overhead.

Other useful built-in variables are the variable **home**, which shows your home directory, **cwd**, which contains your current working directory, and the variable **ignoreeof**, which can be set in your **.login** file to tell the shell not to exit when it receives an end-of-file from a terminal. The variable **ignoreeof** is one of several variables which the shell does not care about the value of, only if it is set or unset. To set this variable you simply do

**set ignoreeof**

and to unset it do

**unset ignoreeof**

These give the variable **ignoreeof** no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables **noclobber** and **mail**. The metasyntax

**> filename**

which redirects the standard output of a command, will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

**set noclobber**

in your .login file. Then trying to do

**date > now**

would cause a diagnostic if 'now' existed already. You could type

**date >! now**

if you really wanted to overwrite the contents of 'now'. The '>!' is a special metasyntax indicating that clobbering the file is OK.

## The History List

The shell can maintain a "history list" into which it places the words of previous commands. It is possible to use a notation to reuse commands or words from previous commands in forming new commands.

The following figures give a sample session involving typical usage of the history mechanism of the shell.

```
% cat bug.c
main()

{
                printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error
% ed !$
ed bug.c
29
4s/);/"&/p
    printf("hello");
w
30
q
%fl !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\\n/p
    printf("hello\n");
w
32
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill     3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill     3932 Dec 19 09:42 bug
% bug
hello
```

```
% num bug.c | spp
spp: Command not found.
% ↑spp↑ssp
num bug.c | ssp
  1             main()
  3             {
  4             printf("hello\n");
  5             }
% !! | lpr
num bug.c | ssp | lpr
%
```

In this example we have a very simple C program which has a bug (or two) in it in the file 'bug.c', which we 'cat' out on our terminal. We then try to run the C compiler on it, referring to the file again as '!$', meaning the last argument to the previous command. Here the '!' is the history metacharacter, and the '$' stands for the last argument.

The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as '!c', which repeats the last command which started with the letter 'c'. If there were other commands starting with 'c' done recently we could have said '!cc' or even '!cc:p' which would have printed the last command starting with 'cc' without executing it.

After this recompilation, we ran the resulting 'a.out' file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra '–o bug' telling the compiler to place the resultant binary in the file 'bug' rather than 'a.out'. In general, the history mechanisms may be used anywhere in the formation of new commands and other characters may be placed before and after the substituted commands.

We then ran the **size** command to see how large the binary program images we have created were, and then an **ls –l** command with the same argument list, denoting the argument list '*'. Finally we ran the program 'bug' to see that its output is indeed correct.

To make a numbered listing of the program we ran the 'num' command on the file 'bug.c'. In order to compress out blank lines in the output of 'num' we ran the output through the filter 'ssp', but misspelled it as spp. To correct this we used a shell substitute, placing the old text and new text between '↑' characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with '!!', but sent its output to the line printer.

There are other mechanisms available for repeating commands. The history command prints out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the C shell manual pages in the *IRIS-4D Programmer's Manual.*

## The Alias Mechanism

The shell has an alias mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility.

As an example, suppose that there is a new version of the mail program on the system called 'newmail' you wish to use, rather than the standard mail program which is called 'mail'. If you place the shell command

**alias mail newmail**

in your **.cshrc** file, the shell will transform an input line of the form

**mail bill**

into a call on 'newmail'. More generally, suppose we wish the command **ls** to always show sizes of files, that is to always do **–s**. We can do

**alias ls ls –s**

or even

**alias dir ls –s**

creating a new command syntax **dir** which does an **ls –s**. If we say

**dir ~bill**

then the shell will translate this to

**ls –s /mnt/bill**

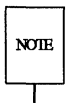
The alias mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

**alias cd 'cd \!* ; ls '**

would do an **ls** command after each **cd** command. We enclosed the entire alias definition in "'" characters to prevent most substitutions from occurring and the character ';' from being recognized as a metacharacter. The '!' is escaped with a '\' to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing **cd** command, without giving an error if there were no arguments. The ';' separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

**alias whois 'grep \!↑ /etc/passwd'**

defines a command which looks up its first argument in the password file.




The shell currently reads the **.cshrc** file each time it starts up. If you place a large number of commands there, shells will tend to start slowly.

## Redirection Commands for Terminal Users

There are a few more notations useful to the terminal user which have not been introduced yet.

In addition to the standard output, commands also have a diagnostic output which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance, if you want to redirect the output of a long-running command into a file, and wish to have a record of any error diagnostic it produces, you can do

**command >& file**

The '>&' here tells the shell to route both the diagnostic output and the standard output into 'file'. Similarly you can give the command

**command | & lpr**

to route both standard and diagnostic output through the pipe to the line printer daemon **lpr**. A command form

command >&! file

exists, and is used when **noclobber** is set and file already exists. Finally, it is possible to use the form

command >> file

to place output at the end of an existing file.

NOTE

If noclobber is set, then an error will result if *file* does not exist, otherwise the shell will create *file* if it doesn't exist. A form command >>! file makes it alright for *file* not to exist when noclobber is set.

## Background, Foreground, or Suspended Jobs

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single job is created by the shell. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are

```
sort < data
ls –s | sort –n | head –5
mail harold
```

If the metacharacter '&' is typed at the end of the commands, then the job is started as a background job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the shell. Thus

**du > usage &**

would run the **du** program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file *usage* and return immediately with a prompt for the next command without waiting for **du** to finish. The **du** program would continue executing in the background until it finished. When a background job terminates, a message is typed by the shell just before the next prompt, telling you that the job has completed.

In the following example the **du** job finishes sometime during the execution of the mail command and its completion is reported just before the prompt after the mail job is finished.

```
% du > usage &
[1] 503
% mail bill
How do you know when a background job is finished?
EOT
[1] – Done                    du > usage
%
```

If the job did not terminate, the 'Done' message might say something else like 'Killed'. If you want the terminations of background jobs to be reported at the time they occur (possibly interrupting the output of other foreground jobs), you can set the notify variable. In the previous example this would mean that the 'Done' message might have come right in the middle of the message to Bill. Background jobs are unaffected by any signals from the keyboard like the stop, interrupt, or quit signals mentioned earlier.

Jobs are recorded in a table inside the shell until they terminate. In this table, the shell remembers the command names, arguments, and the "process numbers" of all commands in the job as well as in the working directory where the job was started. Each job in the table is either running in the foreground with the shell waiting for it to terminate, running in the background, or suspended.

Only one job can be running in the foreground at one time, but several jobs can be suspended or running in the background at once. As each job is started, it is assigned a small identifying number called the "job number" which can be used later to refer to the job in the commands described below. Job numbers remain the same until the job terminates.

When a job is started in the backgound using '&', its number, as well as the process numbers of all its (top-level) commands, is typed by the shell before prompting you for another command. For example,

```
% ls –s | sort –n > usage &
[2] 2034 2035
%
```

runs the **ls** program with the **–s** options, pipes this output into the *sort* program with the **–n** option which puts its output into the file *usage*. Since the '&' was at the end of the line, these two programs were started together as a background job. After starting the job, the shell prints the job number in brackets (2 in this case) followed by the process number of each program started in the job. Then the shell immediately prompts for a new command, leaving the job running simultaneously.

As mentioned before, foreground jobs become suspended by typing ↑Z which sends a stop signal to the currently running foreground job. A background job can become suspended by using the stop command described below. When jobs are suspended they merely stop any further progress until started again, either in the foreground or the backgound. The shell notices when a job becomes stopped and reports this fact, much like it reports the termination of background jobs. For foreground jobs this looks like

```
% du > usage
↑Z
Stopped
%
```

The 'Stopped' message is typed by the shell when it notices that the **du** program stopped. For background jobs, using the **stop** command, it is

```
% sort usage &
[1] 2345
% stop %1
[1] + Stopped (signal)          sort usage
%
```

Suspending foreground jobs can be very useful when you need to temporarily change what you are doing (execute other commands) and then return to the suspended job. Also, foreground jobs can be suspended and then continued as background jobs using the **bg** command, allowing you to continue other work and stop waiting for the foreground job to finish. Thus

```
% du > usage
^Z
Stopped
% bg
[1] du > usage &
%
```

starts **du** in the foreground, stops it before it finishes, then continues it in the background allowing more foreground commands to be executed. This is especially helpful when a foreground job ends up taking longer than you expected.

All job control commands can take an argument that identifies a particular job. All job name arguments begin with the character '%', since some of the job control commands also accept process numbers (printed by the **ps** command.) The default job (when no argument is given) is called the current job and is identified by a '+' in the output which shows you which jobs you have. When only one job is stopped or running in the background (the usual case) it is always the current job and no argument is needed. If a job is stopped while running in the foreground it becomes the current job and the existing current job becomes the previous job identified by a '–' in the output of jobs. When the current job terminates, the previous job becomes the current job. When given, the argument is either '%–' (indicating the previous job); '%#', where # is the job number; '%pref' where pref is some unique prefix of the command name and arguments of one of the jobs; or '%?' followed by some string found in only one of the jobs.

The jobs command types the table of jobs, giving the job number, commands and status ('Stopped' or 'Running') of each backgound or suspended job. With the '–l' option the process numbers are also typed.

```
% du > usage &
[1] 3398
% ls -s | sort -n > myfile &
[2] 3405
% mail bill
^Z
Stopped
% jobs
[1] - Running                 du > usage
[2]   Running                 ls -s | sort -n > myfile
[3] + Stopped                 mail bill
% fg %ls
ls -s | sort -n > myfile
% more myfile
```

The **fg** command runs a suspended or background job in the foreground. It is used to restart a previously suspended job or change a background job to run in the foreground (allowing signals or input from the terminal).

In the above example we used **fg** to change the 'ls' job from the background to the foreground since we wanted to wait for it to finish before looking at its output file.

The **bg** command runs a suspended job in the background. It is usually used after stopping the currently running foreground job with the stop signal. The combination of the stop signal and the **bg** command changes a foreground job into a background job. The stop command suspends a background job.

The **kill** command terminates a background or suspended job immediately. In addition to jobs, it may be given process numbers as arguments, as printed by **ps**. Thus, in the example above, the running **du** command could have been terminated by the command

```
% kill %1
[1]  Terminated                du > usage
%
```

The **notify** command (not the variable mentioned earlier) indicates that the termination of a specific job should be reported at the time it finishes instead of waiting for the next prompt.

If a job running in the background tries to read input from the terminal it is automatically stopped. When such a job is then run in the foreground, input can be given to the job. If desired, the job can be run in the background again until it requests input again. This is illustrated in the following sequence where the 's' command in the text editor might take a long time.

```
% ed bigfile
120000
1,$s/thisword/thatword/
↑Z
Stopped
% bg
[1] ed bigfile &
%
... some foreground commands
[1] Stopped (tty input)            ed bigfile
% fg
ed bigfile
w
120000
q
%
```

So after the **s** command was issued, the **ed** job was stopped with ↑Z and then put in the background using **bg**. Some time later when the **s** command was finished, **ed** tried to read another command and was stopped because jobs in the backgound cannot read from the terminal. The **fg** command returned the **ed** job to the foreground

where it could once again accept commands from the terminal.

The command

**stty tostop**

causes all background jobs run on your terminal to stop when they are about to write output to the terminal. This prevents messages from background jobs from interrupting foreground job output and allows you to run a job in the background without losing terminal output. It also can be used for interactive programs that sometimes have long periods without interaction. Thus, each time it outputs a prompt for more input it will stop before the prompt. It can then be run in the foreground using **fg**, more input can be given and, if necessary stopped and returned to the background. This **stty** command might be a good thing to put in your .login file if you do not like output from background jobs interrupting your work. It also can reduce the need for redirecting the output of background jobs if the output is not very big:

```
% stty tostop
% wc hugefile &
[1] 10387
% ed text
. . . some time later
q
[1] Stopped (tty output)        wc hugefile
% fg wc
wc hugefile
   13371   30123   302577
% stty -tostop
```

Thus after some time the **wc** command, which counts the lines, words and characters in a file, had one line of output. When it tried to write this to the terminal it stopped. By restarting it in the foreground we allowed it to write on the terminal exactly when we were ready to look at its output. Programs which attempt to change the mode of the terminal will also block, whether or not **tostop** is set, when they are not in the foreground, as it would be very unpleasant to have a background job change the state of the terminal.

Since the **jobs** command only prints jobs started in the currently executing shell, it knows nothing about background jobs started in other login sessions or within shell files. The **ps** can be used in this case to find out about background jobs not started in the current shell.

## Working Directories

The shell is always in a particular working directory. The change directory command (**cd**) changes the working directory of the shell, that is, changes the directory you are located in.

It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. The make directory command (**mkdir**) creates a new directory. The print working directory command (**pwd**) reports the absolute pathname of the working directory of the shell, that is, the directory you are located in. Thus in the example below:

```
% pwd
/usr/bill
% mkdir newpaper
% chdir newpaper
% pwd
/usr/bill/newpaper
%
```

the user has created and moved to the directory newpaper where, for example, he might place a group of related files.

No matter where you have moved to in a directory hierarchy, you can return to your 'home' login directory by doing just **cd** with no arguments. The name '..' always means the directory above the current one in the hierarchy, thus

**cd ..**

changes the shell's working directory to the one directly above the current one. The name '..' can be used in any pathname, thus,

**cd ../programs**

means change to the directory 'programs' contained in the directory above the

current one. If you have several directories for different projects under, say, your home directory, this shorthand notation permits you to switch easily between them.

The shell always remembers the pathname of its current working directory in the variable **cwd**. The shell can also be requested to remember the previous directory when you change to a new working directory. If the 'push directory' command **pushd** is used in place of the **cd** command, the shell saves the name of the current working directory on a "directory stack" before changing to the new one. You can see this list at any time by typing the 'directories' command **dirs**.

```
% pushd newpaper/references
/newpaper/references ~
% pushd /usr/lib/tmac
/usr/lib/tmac ~/newpaper/references ~
% dirs
/usr/lib/tmac ~/newpaper/references ~
% popd
~/newpaper/references ~
% popd
~
%
```

The list is printed in a horizontal line, reading left to right, with a tilde (~) as shorthand for your home directory—in this case '/usr/bill'. The directory stack is printed whenever there is more than one entry on it and it changes. It is also printed by a **dirs** command. **dirs** is usually faster and more informative than **pwd** since it shows the current working directory as well as any other directories remembered in the stack.

The **pushd** command with no argument alternates the current directory with the first directory in the list. The 'pop directory' (**popd**) command without an argument returns you to the directory you were in prior to the current one, discarding the previous current directory from the stack (forgetting it). Typing **popd** several times in a series takes you backward through the directories you had been in (changed to) by **pushd** command. There are other options to **pushd** and **popd** to manipulate the contents of the directory stack and to change to directories not at the top of the stack; see the **csh** manual page for details.

Since the shell remembers the working directory in which each job was started, it warns you when you might be confused by restarting a job in the foreground which has a different working directory than the current working directory of the shell. Thus if you start a background job, then change the shell's working directory and then cause the background job to run in the foreground, the shell warns you that the working directory of the currently running foreground job is different from that of the shell.

```
% dirs -l
/mnt/bill
% cd myproject
% dirs
~/myproject
% ed prog.c
1143
^Z
Stopped
% cd ..
% ls
myproject
textfile
% fg
ed prog.c (wd: ~/myproject)
```

This way the shell warns you when there is an implied change of working directory, even though no **cd** command was issued. In the above example the 'ed' job was still in **/mnt/bill/project** even though the shell had changed to **/mnt/bill**. A similar warning is given when such a foreground job terminates or is suspended (using the stop signal) since the return to the shell again implies a change of working directory.

```
% fg
ed prog.c (wd: ~/myproject)
 . . . after some editing
q
(wd now: ~)
%
```

These messages are sometimes confusing if you use programs that change their own working directories, since the shell only remembers which directory a job is started in, and assumes it stays there. The –l option of jobs will type the working directory of suspended or background jobs when it is different from the current working directory of the shell.

## Useful Built-in Commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The **alias** command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

**alias ls**

to show the current alias for, e.g., **ls**.

The **echo** command prints its arguments. It is often used in shell scripts or as an interactive command to see what filename expansions will produce.

The **history** command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called **prompt**. By placing a '!' character in its value the shell will there substitute the number of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

set prompt='\! % '

Note that the '!' character had to be escaped here even within '' characters.

The **limit** command is used to restrict use of resources. With no arguments it prints the current limitations:

```
cputime         unlimited
filesize        unlimited
datasize        5616 kbytes
stacksize       512 kbytes
coredumpsize    unlimited
```

Limits can be set, e.g.:

limit coredumpsize 128k

Most reasonable units abbreviations will work; see the **csh** manual page for more details.

The **logout** command can be used to terminate a login shell which has **ignoreeof** set.

The **rehash** command causes the shell to recompute a table to direct you to where commands are located. This is necessary if you add a command to a directory in the current shell's search path and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn't in that directory when the hash table was computed.

The **repeat** command can be used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could do

**repeat 5 cat** *one* **>>** *five*

The **setenv** command can be used to set variables in the environment. Thus

**setenv TERM adm3a**

will set the value of the environment variable TERM to 'adm3a'. A user program **printenv** exists which will print out the environment. It might then show:

```
% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
```

The **source** command can be used to force the current shell to read commands from a file. Thus

**source .cshrc**

can be used after editing in a change to the **.cshrc** file which you wish to take effect before the next time you login.

The **time** command can be used to cause a command to be timed no matter how much time it takes. Thus

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8% 2+1k 3+2io 1pf+0w
% time wc /etc/rc /usr/bill/rc
      52     178    1347 /etc/rc
      52     178    1347 /usr/bill/rc
     104     356    2694 total
0.1u 0.1s 0:00 13% 3+3k 5+3io 7pf+0w
%
```

indicates that the **cp** command used a negligible amount of user time (u) and about 1/10th of a system time (s); the elapsed time was 1 second (0:01), there was an average memory usage of 2K bytes of program space and 1K bytes of data space over the cpu time involved (2+1K); the program did three disk reads and two disk writes (3+2io), and took one page fault and was not swapped (1pf+0w). The word count command **wc** on the other hand used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage '13%' indicates that over the period when it was active the command **wc** used an average of 13 percent of the available cpu cycles of the machine.

The **unalias** and **unset** commands can be used to remove aliases and variable definitions from the shell, and **unsetenv** removes variables from the environment.

## Additional Information

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the **foreach** built-in command which can be used to run the same command sequence with a number of different arguments.

# Executing Commands Through Shell Scripts

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called "shell scripts." We here detail those features of the shell useful to the writers of such scripts.

## The Make Program

It is important to first note what shell scripts are not useful for. There is a program called **make** which is useful for maintaining a group of related files or performing sets of operations on related files. For instance, a large program consisting of one or more files can have its dependencies described in a **Makefile** which contains definitions of the commands used to create these different files when changes occur.

Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately, placed in this **Makefile**. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a **Makefile** may be created which defines how different versions of the document are to be created and which options of **nroff** or **troff** are appropriate.

## Invocation and the argv Variable

A **csh** command script may be interpreted by saying

```
% csh script ...
```

where *script* is the name of the file containing a group of **csh** commands and '...' is replaced by a sequence of arguments. The shell places these arguments in the variable **argv** and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file *script* executable by doing

```
% chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a '#' character) then a **/bin/csh** will automatically be invoked to execute **script** when you type

```
% script
```

If the file does not begin with a '#' then the standard shell **/bin/sh** will be used to execute it. This allows you to convert your older shell scripts to use **csh** at your convenience.

## Substituting Variables

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as variable substitution is done on these words. Keyed by the character **$** this substitution replaces the names of variables by their values. Thus

```
% echo $argv
```

when placed in a command script would cause the current value of the variable **argv** to be echoed to the output of the shell script. It is an error for **argv** to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
% $?name
```

expands to '1' if name is set or to '0' if name is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable name. Thus

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

**$argv[1]**

gives the first component of **argv** or in the example above 'a'. Similarly

**$argv[$#argv]**

would give *c*, and

**$argv[1–2]**

would give **a b**. Other notations useful in shell scripts are

**$*n***

where *n* is an integer as a shorthand for

$argv[*n*]

the **n th** parameter and

**$***

which is a shorthand for

**$argv**

The form $$ expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names. The form

**$<**

is quite special and is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
echo 'yes or no?\c'
set a=($<)
```

would write out the prompt 'yes or no?' without a newline and then read the answer into the variable **a**. In this case **$#a** would be **0** if either a blank line or end-of-file (↑D) was typed.

One minor difference between $*n* and **$argv**[*n*] should be noted here. The form **$argv**[*n*] will yield an error if **n** is not in the range **1–$#argv** while **$n** will never yield an out-of-range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form **n–**; if there are less than **n** components of the given variable then no words are substituted. A range of the form **m–n** likewise returns an empty vector without giving an error when **m** exceeds the number of elements of the given variable, provided the subscript **n** is in range.

# Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations == and != compare strings and the operators && and | | implement the boolean and/or operations. The special operators =~ and !~ are similar to == and != except that the string on the right side can have pattern matching characters (like *, ? or []) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

**–? filename**

where '?' is replaced by a number of single characters. For instance the expression primitive

**–e** *filename*

tells whether the file *filename* exists. Other primitives test for read, write, and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form '{ command }' which returns true, i.e. '1' if the command succeeds exiting normally with exit status 0, or '0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable **$status** examined in the next command. Since **$status** is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in an immediately following single command.

For a full list of expression components available see the *IRIS-4D Programmer's Reference Manual.*

## Sample Shell Script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

    if ($i !~ *.c) continue  # not a .c file so do nothing

    if (! -r ~/backup/$i:t) then
        echo $i:t not in backup... not cp\'ed
        continue
    endif

    cmp -s $i ~/backup/$i:t # to set $status

    if ($status != 0) then
        echo new backup of $i
        cp $i ~/backup/$i:t
    endif
end
```

This script makes use of the **foreach** command, which causes the shell to execute the commands between the **foreach** and the matching end for each of the values given between '(' and ')' with the named variable, in this case **i** set to successive values in the list. Within this loop we may use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop the iteration variable (**i** in this case) has the value at the last iteration.

We set the variable **noglob** here to prevent filename expansion of the members of **argv**. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a '$' variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

```
if ( expression ) then
                        command
                        ...
endif
```

The placement of the keywords here is **not** flexible due to the current implementation of the shell. The following two formats are not currently acceptable to the shell:

```
if ( expression )       # Won't work!
then
                        command
                        ...
endif
```

and

```
if ( expression ) then command endif# Won't work
```

The shell does have another form of the if statement of the form

```
if ( expression ) command
```

which can be written

```
if ( expression ) \
                        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve '|', '&' or ';' and must not be another control command. The second form requires the final '\' to immediately precede the end-of-line.

The more general 'if' statements above also admit a sequence of else–if pairs followed by a single 'else' and an 'endif,' e.g.:

```
if ( expression ) then
                        commands
else if (expression ) then
                        commands
...
else
                        commands
endif
```

Another important mechanism used in shell scripts is the : modifier. We can use the modifier **:r** here to extract a root of a filename or **:e'** to extract the extension. Thus if the variable **i** has the value '/mnt/foo.bar' then

```
% echo $i $i:r $i:e
/mnt/foo.bar /mnt/foo bar
%
```

shows how the **:r** modifier strips off the trailing '.bar' and the the **:e** modifier leaves only the 'bar'. Other modifiers will take off the last component of a pathname leaving the head ':h' or all but the last component of a pathname leaving the tail ':t'. These modifiers are fully described in the **csh** manual pages in the *IRIS-4D Programmer's Manual.* It is also possible to use the command substitution mechanism described in the next section to perform modifications on strings to then reenter the shells environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the : modification mechanism.

It is also important to note that the current implementation of the shell limits the number of : modifiers on a '$' substitution to 1. Thus

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

does not do what one would expect.

Finally, we note that the character '#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a '#' are discarded by the shell. This character can be quoted using '' or '\' to place it in an argument word.

## Other Control Structures

The shell also has control structures **while** and **switch** similar to those of C. These take the forms

```
while ( expression )
                        commands
end
```

and

```
switch ( word )

case str1:
                        commands
                        breaksw

...

case strn:
                        commands
                        breaksw

default:
                        commands
                        breaksw

endsw
```

For details see the manual section for **csh**. C programmers should note that we use **breaksw** to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in **csh** scripts is to use **break** rather than **breaksw** in switches.

Finally, **csh** allows a **goto** statement, with labels looking like they do in C, i.e.:

```
loop:
                        commands
                        goto loop
```

## Supplying Input to Commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This is different from previous shells running under UNIX. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank — remove leading blanks
foreach i ($argv)
ed – $i << 'EOF'
1,$s/↑[ ]*//
w
q
'EOF'
end
%
```

The notation '<< 'EOF'' means that the standard input for the **ed** command is to come from the text in the shell script file up to the next line consisting of exactly ''EOF''. The fact that the 'EOF' is enclosed in '' characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,$' in our editor script we needed to insure that this '$' was not variable substituted. We could also have insured this by preceding the '$' here with a '\', i.e.:

```
1,\$s/↑[ ]*//
```

but quoting the 'EOF' terminator is a more reliable way of achieving the same thing.

## Catching Interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where label is a label in our program. If an interrupt is received the shell will do a 'goto label' and we can remove the temporary files and then do an **exit** command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status '1'.

## Other Shell Features

There are other features of the shell useful to writers of shell procedures. The **verbose** and **echo** options and the related **–v** and **–x** command line options can be used to help trace the actions of the shell. The **–n** option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that **csh** will not execute shell scripts which do not begin with the character '#', that is shell scripts that do not begin with a comment. Similarly, the **/bin/sh** on your system may well defer to **csh** to interpret shell scripts which begin with '#'. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using '"' which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as '`' does.

## Loops at the Terminal; Variables as Vectors

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, **/bin/sh**, **/bin/nsh**, and **/bin/csh**. To count the number of persons using each shell one could have issued the commands

```
% grep –c csh$ /etc/passwd
27
% grep –c nsh$ /etc/passwd
128
% grep –c –v sh$ /etc/passwd
430
%
```

Since these commands are very similar we can use **foreach** to do this more easily.

```
% foreach i (´sh$´ ´csh$´ ´–v sh$´)
? grep –c $i /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with '? ' when reading the body of the loop.

Variables which contain lists of filenames or other words are useful with loops. You can, for example, do

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The **set** command here gave the variable **a** as value a list of all the filenames in the current directory. We can then iterate over these names to perform any chosen function.

The output of a command within '`' characters is converted by the shell to a list of words. You can also place the '`' quoted string within '"' characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier ':x' exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

## Braces { ... } in Argument Expansion

Another form of filename expansion involves the characters '{' and '}'. These characters specify that the contained strings, separated by ',' are to be consecutively substituted into the braces and the results expanded left to right. Thus

A{str1,str2,...strn}B

expands to

Astr1B Astr2B ... AstrnB

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be

used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

**mkdir ~/{hdrs,retrofit,csh}**

to make subdirectories **hdrs**, **retrofit**, and **csh** in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

**chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}**

## Command Substitution

A command enclosed in "`" characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

**set pwd=`pwd`**

to save the current directory in the variable **pwd** or to do

**ex `grep –l TRACE *.c`**

to run the editor **ex** supplying as arguments those files whose names end in '.c' which have the string 'TRACE' in them.

Command expansion also occurs in input redirected with '<<' and within '"' quotations. Refer to the *IRIS-4D Programmer's Manual* for full details.

## Additional Information

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in the manual sections.

The shell has a number of command line option flags you can use to write UNIX programs and debug shell scripts. See the shell's manual section for a list of these options.

# C Shell Special Characters

The following table lists the special characters of **csh** and the UNIX system. A number of these characters also have special meaning in expressions. See the **csh** manual section for a complete list. Syntactic metacharacters

| | |
|---|---|
| ; | separates commands to be executed sequentially |
| \| | separates commands in a pipeline |
| ( ) | brackets expressions and variable values |
| & | follows commands to be executed without waiting for completion |

Filename metacharacters

| | |
|---|---|
| / | separates components of a file's pathname |
| ? | expansion character matching any single character |
| * | expansion character matching any sequence of characters |
| [ ] | expansion sequence matching any single character from a set |
| ~ | used at the beginning of a filename to indicate home directories |
| { } | used to specify groups of arguments with common parts |

Quotation metacharacters

| | |
|---|---|
| \ | prevents meta-meaning of following single character |
| ' | prevents meta-meaning of a group of characters |
| " | like ', but allows variable and command expansion |

Input/output metacharacters

| | |
|---|---|
| < | indicates redirected input |
| > | indicates redirected output |

Expansion/substitution metacharacters

| | |
|---|---|
| $ | indicates variable substitution |
| ! | indicates history substitution |
| : | precedes substitution modifiers |
| ↑ | used in special forms of history substitution |
| ` | indicates command substitution |

Other metacharacters

| | |
|---|---|
| # | begins scratch file names; indicates shell comments |
| – | prefixes option (flag) arguments to commands |
| % | prefixes job name specifications |

# C Shell Glossary

This glossary lists the most important terms introduced in Chapter 8 and gives references to sections of the chapter for further information about them. References of the form **pr (1)** indicate that the command **pr** is in the *UNIX Programmer's Manual* in section 1. You can get an on-line copy of its manual page by doing

**man 1 pr**

Your current directory has the name '.' as well as the name printed by the command **pwd**; see also **dirs.** The current directory '.' is usually the first component of the search path contained in the variable path, thus commands which are in '.' are found first. The character '.' is also used in separating components of filenames. The character '.' at the beginning of a component of a pathname is treated specially and not matched by the file-name expansion metacharacters '?', '*', and '[' ']' pairs.

**..** Each directory has a file '..' in it which is a reference to its parent directory. After changing into the directory with **chdir**, i.e.:

**chdir paper**

you can return to the parent directory by doing

**chdir ..**

The current directory is printed by **pwd.**

**a.out**
Compilers which create executable images create them, by default, in the file **a.out.**.

**absolute pathname**
A pathname which begins with a '/' is absolute since it specifies the path of directories from the beginning of the entire directory system – called the root directory. Pathnames which are not absolute are called relative (see definition of relative pathname ).

**alias**
An alias specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command alias which establishes aliases and can print their current values. The command unalias is used to remove aliases.

**argument**
Commands in UNIX receive a list of argument words. Thus the command

**echo a b c**

consists of the command name 'echo' and three argument words 'a', 'b' and 'c'. The set of arguments after the command name is said to be the argument list of the command.

**argv**

The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called **argv** within the shell. This name is taken from the conventional name in the C programming language.

**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.

**base**

A file name is sometimes thought of as consisting of a base part, before any '.' character, and an extension – the part after the '.'. See filename and extension.

**bg**

The bg command causes a suspended job to continue execution in the background.

**bin**

A directory containing binaries of programs and shell scripts to be executed is typically called a bin directory. The standard system bin directories are /bin which contains the most heavily used commands and /usr/bin which contains most other user programs. Programs developed at UC Berkeley live in /usr/ucb, while locally written programs live in /usr/local. Games are kept in the directory /usr/games. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a component of the variable path.

**break**

Break is a built-in command used to exit from loops within the control structure of the shell.

**breaksw**

The breaksw built-in command is used to exit from a switch control structure, like a break exits from loops.

**builtin**

A command executed directly by the shell is called a built-in command. Most commands in UNIX are not built into the shell, but rather exist as files in bin directories. These commands are accessible because the

directories in which they reside are named in the path variable.

**case**
A case command is used as a label in a switch statement in the shell's control structure, similar to that of the language C. Details are given in the shell documentation 'csh(1)'.

**cat**
The cat program catenates a list of specified files on the standard output. It is usually used to look at the contents of a single file on the terminal, to 'cat a file'.

**cd**
The cd command is used to change the working directory. With no arguments, cd changes your working directory to be your home directory.

**chdir**
The chdir command is a synonym for **cd**. **cd** is usually used because it is easier to type.

**chsh**
The **chsh** command is used to change the shell which you use on UNIX. By default, you use an different version of the shell which resides in /bin/sh. You can change your shell to /bin/csh by doing

**chsh your-login-name /bin/csh**

Thus I would do

**chsh bill /bin/csh**

It is only necessary to do this once.
The next time you log in to UNIX after doing this command, you will be using csh rather than the shell in /bin/sh.

**cmp**
cmp is a program which compares files. It is usually used on binary files, or to see if two files are identical. For comparing text files the program diff, described in diff (1) is used.

**command**
A function performed by the system, either by the shell (a built-in command ) or by a program residing in a file in a directory within the UNIX system, is called a command.

**command name**
When a command is issued, it consists of a command name, which is the first word of the command, followed by arguments. The convention on UNIX is that the first word of a command names the function to be

performed.

**command substitution**
The replacement of a command enclosed in '`' characters by the text output by that command is called command substitution.

**component**
A part of a pathname between '/' characters is called a component of that pathname. A variable which has multiple strings as value is said to have several components; each string is a component of the variable.

**continue**
A built-in command which causes execution of the enclosing foreach or while loop to cycle prematurely. Similar to the continue command in the programming language C.

**ctrl-**
Certain special characters, called control characters, are produced by holding down the control key on your terminal and simultaneously pressing another character, much like the shift key is used to produce uppercase characters. Thus ctrl-C is produced by holding down the control key while pressing the 'C' key.

**core dump**
When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This "core dump" can be examined with the system debugger 'adb(1)' or 'sdb(1)' in order to determine what went wrong with the program. If the shell produces a message of the form

**Illegal instruction (core dumped)**

(where Illegal instruction is only one of several possible messages), you should report this to the author of the program or a system administrator, saving the 'core' file.

**cp**
The cp (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used UNIX commands).

**csh**
The name of the shell program that this document describes.

**.cshrc**
The file .cshrc in your home directory is read by each shell as it begins execution. It is usually used to change the setting of the variable path and to set alias parameters which are to take effect globally.

**cwd**

The cwd variable in the shell holds the absolute pathname of the current working directory . It is changed by the shell whenever your current working directory changes and should not be changed otherwise.

**debugging**

Debugging is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell debugging.

**default**

The label default is used within shell switch statements, as it is in the C language to label the code to be executed if none of the case labels matches the value switched on.

**delete**

The delete or rubout key on the terminal normally causes an interrupt to be sent to the current job. Many users change the interrupt character to be ctrl-C.

**detached**

A command that continues running in the background after you logout is said to be detached.

**diagnostic**

An error message produced by a program is often referred to as a diagnostic. Most error messages are not written to the standard output, since that is often directed away from the terminal. Error messsages are instead written to the diagnostic output which may be directed away from the terminal, but usually is not. Thus diagnostics will usually appear on the terminal.

**directory**

A structure which contains files. At any time you are in one particular directory whose names can be printed by the command **pwd**. The chdir command will change you to another directory, and make the files in that directory visible. The directory in which you are when you first login is your home directory.

**directory stack**

The shell saves the names of previous working directories in the directory stack when you change your current working directory via the **pushd** command. The directory stack can be printed by using the **dirs** command, which includes your current working directory as the first directory name on the left.

**dirs**

The dirs command prints the shell's directory stack.

**du**

The du command is a program (described in 'du(1)') which prints the number of disk blocks is all directories below and including your current working directory.

**echo**

The **echo** command prints its arguments.

**else**

The else command is part of the 'if-then-else-endif' control command construct.

**endif**

If an 'if' statement is ended with the word 'then', all lines following the if up to a line starting with the word endif or else are executed if the condition between parentheses after the if is true.

**EOF**

An end-of-file is generated by the terminal by a ctrl-d, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a pipe receive an "end-of-file" when the command sending them input completes. Most commands terminate when they receive an end-of-file. The shell has an option to ignore end-of-file from a terminal input which may help you keep from logging out accidentally by typing too many ctrl-d's.

**escape**

A character '\' used to prevent the special meaning of a metacharacter is said to escape the character from its special meaning. Thus:

```
echo \*
```

will echo the character '*' while just

```
echo *
```

will echo the names of the file in the current directory.
In this example, \ escapes '*'.
There is also a non-printing character called escape,
usually labelled escap or altmode on terminal keyboards.
Some older UNIX systems use this character to indicate that
output is to be suspended.
Most systems use ctrl-s to stop the output and ctrl-q to start it.

**/etc/passwd**

This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by ':' characters. You can look at this file by saying:

```
cat /etc/passwd
```

The commands **finger** and **grep**
are often used to search for information in this file.
See 'finger(1)', 'passwd(5)', and 'grep(1)' for more details.

**exit**

The exit command is used to force termination of a shell script, and is built into the shell.

**exit status**

A command which discovers a problem may reflect this back to the command (such as a shell) which invoked (executed) it. It does this by returning a non-zero number as its exit status, a status of zero being considered normal termination. The exit command can be used to force a shell command script to give a non-zero exit status.

**expansion**

The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of expansion. Thus the replacement of the word '*' by a sorted list of files in the current directory is a filename expansion. Similarly the replacement of the characters '!!' by the text of the last command is a history expansion. Expansions are also referred to as substitutions.

**expressions**

Expressions are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. are those of the language C.

**"extension"**

Filenames often consist of a base name and an extension separated by the character '.'. By convention, groups of related files often share the same root name. Thus if prog.c were a C program, then the object file for this program would be stored in prog.o. Similarly a paper written with the –me nroff macro package might be stored in paper.me while a formatted version of this paper might be kept in paper.out and a list of spelling errors in paper.errs.

**fg**

The job control command fg is used to run a background or suspended job in the foreground.

**filename**

Each file in UNIX has a name consisting of up to 14 characters and not including the character '/' which is used in pathname building. Most filenames do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the base portion of the filename from an extension.

**filename expansion**

Filename expansion uses the metacharacters '*', '?' and '[' and ']' to provide a convenient mechanism for naming files. Using filename expansion it is easy to name all the files in the current directory, or all files which have a common root name. Other filename expansion mechanisms use the metacharacter '~' and allow files in other users' directories to be named easily.

**flag**

Many UNIX commands accept arguments which are not the names of files or other users but are used to modify the action of the commands. These are referred to as **flag** options, and by convention consist of one or more letters preceded by the character '–'. Thus the **ls** (list files) command has an option '–s' to list the sizes of files. This is specified

```
ls –s
```

**foreach**

The foreach command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list.

**foreground**

When commands are executing in the normal way such that the shell is waiting for them to finish before prompting for another command they are said to be foreground jobs or running in the foreground. Foreground jobs can be stopped by signals from the terminal caused by typing different control characters at the keyboard.

**goto**

The shell has a command goto used in shell scripts to transfer control to a given label.

**grep**

The grep command searches through a list of argument files for a specified string. Thus

```
grep bill /etc/passwd
```

will print each line in the file /etc/passwd which contains the string 'bill'.

Actually, grep scans for regular expressions in the sense of the editors ed(1) and ex(1). grep stands for 'globally find regular expression and print'.

**head**

The head command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes useful to run head with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in. Head is also used to describe the part of a pathname before and including the last '/' character. The tail of a pathname is the part after the last '/'. The ':h' and ':t' modifiers allow the head or tail of a pathname stored in a shell variable to be used.

**history**

The history mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a history list where these commands are kept, and a history variable which controls how large this list is.

**home directory**

Each user has a home directory, which is given in your entry in the password file, /etc/passwd. This is the directory which you are placed in when you first login. The **cd** or **chdir** command with no arguments takes you back to this directory, whose name is recorded in the shell variable home. You can also access the home directories of other users in forming filenames using a filename expansion notation and the character '~'.

**if**

A conditional command within the shell, the if command is used in shell command scripts to make decisions about what course of action to take next.

**ignoreeof**

Normally, your shell will exit, printing 'logout' if you type a ctrl-d at a prompt of '%'. This is the way you usually log off the system. You can set the ignoreeof variable if you wish in your .login file and then use the command logout to logout. This is useful if you sometimes accidentally type too many ctrl-d characters, logging yourself off.

**input**

Many commands on UNIX take information from the terminal or from files which they then act on. This information is called input. Commands normally read for input from their standard input which is, by default, the terminal. This standard input can be redirected from a file using a shell metanotation with the character '<'. Many commands will also read from a file specified as argument. Commands placed in

pipelines will read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if you neither redirect its input nor give it a filename to use as standard input. Special mechanisms exist for supplying input to commands in shell scripts.

**interrupt**
An interrupt is a signal to a program that is generated by hitting the rubout or delete key (although users can and often do change the interrupt character, usually to ctrl-C. It causes most programs to stop execution. Certain programs, such as the shell and the editors, handle an interrupt in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to interrupts. The shell often wakes up when you hit interrupt because many commands die when they receive an interrupt.

**job**
One or more commands typed on the same input line separated by '|' or ';' characters are run together and are called a job . Simple commands run by themselves without any '|' or ';' characters are the simplest jobs. Jobs are classified as foreground, background, or suspended.

**job control**
The built-in functions that control the execution of jobs are called job control commands. These are bg, fg, stop, and kill.

**job number**
When each job is started it is assigned a small number called a job number which is printed next to the job in the output of the jobs command. This number, preceded by a % character, can be used as an argument to job control commands to indicate a specific job.

**jobs**
The jobs command prints a table showing jobs that are either running in the background or are suspended.

**kill**
A command which sends a signal to a job causing it to terminate.

**.login**
The file .login in your home directory is read by the shell each time you login to UNIX and the commands there are executed. There are a number of commands which are usefully placed here, especially set commands to the shell itself.

**login shell**
The shell that is started on your terminal when you login is called your login shell. It is different from other shells which you may run (e.g. on shell scripts) in that it reads the .login file before reading commands from the terminal and it reads the .logout file after you logout.

**logout**
The logout command causes a login shell to exit. Normally, a login shell will exit when you hit ctrl-d generating an end-of-file, but if you have set ignoreeof in your .login file then this will not work and you must use logout to log off the UNIX system.

**.logout**
When you log off of UNIX the shell will execute commands from the file .logout in your home directory after it prints logout.

**lpr**
The command **lpr** is the line printer daemon. The standard input of lpr spooled and printed on the UNIX line printer. You can also give lpr a list of filenames as arguments to be printed. It is most common to use lpr as the last component of a pipeline.

**ls**
The ls command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful flag arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories.

**mail**
The mail program is used to send and receive messages from other UNIX users.

**make**
The make command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways make is easier to use, and more helpful than shell command scripts.

**makefile**
The file containing commands for make is called makefile.

**manual**
The manual often referred to is the *UNIX Programmer's Manual.* It contains a number of sections and a description of each UNIX program. An on-line version of the manual is accessible through the man command. Its documentation can be obtained on-line via

```
man man
```

**metacharacter**
Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called metacharacters. If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be quoted. An example of a metacharacter is the character '>' which is used to indicate placement of output into a file. For the purposes of the history mechanism, most unquoted metacharacters form separate words. The appendix to this user's manual lists the metacharacters in groups by their function.

**mkdir**
The mkdir command is used to create a new directory.

**modifier**
Substitutions with the history mechanism, keyed by the character '!' or of variables using the metacharacter '$', are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the modifier itself. The command substitution mechanism can also be used to perform modification in a similar way, but this notation is less clear.

**more**
The program more writes a file on your terminal allowing you to control how much text is displayed at a time. More can move through the file screenful by screenful, line by line, search forward for a string, or start again at the beginning of the file. It is generally the easiest way of viewing a file.

**noclobber**
The shell has a variable noclobber which may be set in the file .login to prevent accidental destruction of files by the '>' output redirection metasyntax of the shell.

**noglob**
The shell variable noglob is set to suppress the filename expansion of arguments containing the metacharacters '~', '*', '?', '[' and ']'.

**notify**
The notify command tells the shell to report on the termination of a specific background job at the exact time it occurs as opposed to waiting until just before the next prompt to report the termination. The notify variable, if set, causes the shell to always report the termination of background jobs exactly when they occur.

**onintr**

The onintr command is built into the shell and is used to control the action of a shell command script when an interrupt signal is received.

**output**

Many commands in UNIX result in some lines of text which are called their output. This output is usually placed on what is known as the standard output which is normally connected to the user's terminal. The shell has a syntax using the metacharacter '>' for redirecting the standard output of a command to a file. Using the pipe mechanism and the metacharacter '|' it is also possible for the standard output of one command to become the standard input of another command.

Certain commands such as the line printer daemon **p** do not place their results on the standard output but rather in more useful places such as on the line printer. Similarly the write command places its output on another user's terminal rather than its standard output.

Commands also have a diagnostic output where they write their error messages. Normally these go to the terminal even if the standard output has been sent to a file or another command, but it is possible to direct error diagnostics along with standard output using a special metanotation.

**pushd**

The **pushd** command, which means push directory, changes the shell's working directory and also remembers the current working directory before the change is made, allowing you to return to the same directory via the **popd** command later without retyping its name.

**path**

The shell has a variable **path** which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the shell. If it is, then it need not search for the command as it can do it internally. If the command is not built-in, then the shell searches for a file with the name given in each of the directories in the path variable, left to right. Since the normal definition of the path variable is

**path (. /usr/ucb /bin /usr/bin)**

the shell normally looks in the current directory, and then in the standard system directories /usr/ucb, /bin, and /usr/bin for the named command. If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have execute permission set. This is normally true

because a command of the form

**chmod 755 script**

was executed to turn this execute permission on.
If you add new commands to a directory in the path, you should issue the command rehash.

**pathname**
A list of names, separated by '/' characters, forms a pathname. Each component, between successive '/' characters, names a directory in which the next component file resides. Pathnames which begin with the character '/' are interpreted relative to the root directory in the file system. Other pathnames are interpreted relative to the current directory as reported by **pwd**. The last component of a pathname may name a directory, but usually names a file.

**pipeline**
A group of commands which are connected together, the standard output of each connected to the standard input of the next, is called a pipeline. The pipe mechanism used to connect these commands is indicated by the shell metacharacter '|'.

**popd**
The **popd** command changes the shell's working directory to the directory you most recently left using the **pushd** command. It returns to the directory without having to type its name, forgetting the name of the current working directory before doing so.

**port**
The part of a computer system to which each terminal is connected is called a port. Usually the system has a fixed number of ports , some of which are connected to telephone lines for dial-up access, and some of which are permanently wired directly to specific terminals.

**pr**
The **pr** command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified.

**printenv**
The **printenv** command is used to print the current setting of variables in the environment.

**process**
An instance of a running program is called a process. UNIX assigns each process a unique number when it is started – called the process number. Process numbers can be used to stop individual processes using

the kill or stop commands when the processes are part of a detached background job.

**program**
Usually synonymous with command; a binary file or shell command script which performs a useful function is often called a program.

**prompt**
Many programs will print a prompt on the terminal when they expect input. Thus the editor ex(1) will print a ':' when it expects input. The shell prompts for input with % and occasionally with ? when reading commands from the terminal. The shell has a variable prompt which may be set to a different value to change the shell's main prompt. This is mostly used when debugging the shell.

**ps**
The ps command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, an indication of the state of the process (whether it is running, stopped, awaiting some event (sleeping), and whether it is swapped out), and the amount of cpu time it has used so far. The command is identified by printing some of the words used when it was invoked. Shells, such as the **csh** you use to run the **ps** command, are not normally shown in the output.

**pwd**
The pwd command prints the full pathname of the current working directory. The dirs built-in command is usually a better and faster choice.

**quit**
The quit signal, generated by a ctrl-e, is used to terminate programs which are behaving unreasonably. It normally produces a core image file.

**quotation**
The process by which metacharacters are prevented their special meaning, usually by using the character ' in pairs, or by using the character '\', is referred to as quotation.

**redirection**
The routing of input or output from or to a file is known as redirection of input or output.

**rehash**
The rehash command tells the shell to rebuild its internal table of which commands are found in which directories in your path. This is necessary when a new program is installed in one of these directories.

**relative pathname**

A pathname which does not begin with a '/' is called a relative pathname since it is interpreted relative to the current working directory. The first component of such a pathname refers to some file or directory in the working directory, and subsequent components between '/' characters refer to directories below the working directory. Pathnames that are not relative are called absolute pathnames.

**repeat**

The repeat command iterates another command a specified number of times.

**root**

The directory that is at the top of the entire directory structure is called the root directory since it is the 'root' of the entire tree structure of directories. The name used in pathnames to indicate the root is '/'. Pathnames starting with '/' are said to be absolute since they start at the root directory. Root is also used as the part of a pathname that is left after removing the extension. See filename for a further explanation.

**rubout**

The rubout or delete key sends an interrupt to the current job. Most interactive commands return to their command level upon receipt of an interrupt, while non-interactive commands usually terminate, returning control to the shell. Users often change interrupt to be generated by ctrl-C rather than delete by using the stty command.

**scratch file**

Files whose names begin with a '#' are referred to as scratch files, since they are automatically removed by the system after a couple of days of non-use, or more frequently if disk space becomes tight.

**script**

Sequences of shell commands placed in a file are called shell command scripts. It is often possible to perform simple tasks using these scripts without writing a program in a language such as C, by using the shell to selectively run other programs.

**set**

The built-in set command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the set command the behavior of the shell can be affected.

**setenv**
Variables in the environment environ(5) can be changed by using the setenv built-in command. The printenv command can be used to print the value of the variables in the environment.

**shell**
A shell is a command language interpreter. It is possible to write and run your own shell, as shells are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular shell, called csh.

**shell script**
See script.

**signal**
A signal in UNIX is a short message that is sent to a running program which causes something to happen to that process. Signals are sent either by typing special control characters on the keyboard or by using the kill or stop commands.

**sort**
The sort program sorts a sequence of lines in ways that can be controlled by argument flags.

**source**
The source command causes the shell to read commands from a specified file. It is most useful for reading files such as .cshrc after changing them.

**special character**
See metacharacters.

**standard**
We refer often to the standard input and standard output of commands. See input and output.

**status**
A command normally returns a status when it finishes. By convention a status of zero indicates that the command succeeded. Commands may return non-zero status to indicate that some abnormal event has occurred. The shell variable status is set to the status returned by the last command. It is most useful in shell commmand scripts.

**stop**
The stop command causes a background job to become suspended.

**string**
A sequential group of characters taken together is called a string . Strings can contain any printable characters.

**stty**
The stty program changes certain parameters inside UNIX which determine how your terminal is handled. See stty(1) for a complete description.

**substitution**
The shell implements a number of substitutions where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history substitution keyed by the metacharacter '!' and variable substitution indicated by '$'. We also refer to substitutions as expansions.

**suspended**
A job becomes suspended after a stop signal is sent to it, either by typing a ctrl-z at the terminal (for foreground jobs) or by using the stop command (for background jobs). When suspended , a job temporarily stops running until it is restarted by either the **fg** or **bg** command.

**switch**
The switch command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the switch statement in the language C.

**termination**
When a command which is being executed finishes we say it undergoes termination or terminates. Commands normally terminate when they read an end-of-file from their standard input. It is also possible to terminate commands by sending them an interrupt or quit signal. The kill program terminates specified jobs.

**then**
The then command is part of the shell's if-then-else-endif control construct used in command scripts.

**time**
The time command can be used to measure the amount of cpu and real time consumed by a specified command as well as the amount of disk I/O, memory utilized, and number of page faults and swaps taken by the command.

**tset**
The tset program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a .login file.

**tty**

tty is a historical abbreviation for 'teletype' which is frequently used in UNIX to indicate the port to which a given terminal is connected. The tty command will print the name of the tty or port to which your terminal is presently connected.

**unalias**

The unalias command removes aliases.

**UNIX**

UNIX is an operating system on which **csh** runs. UNIX provides facilities which allow **csh** to invoke other programs such as editors and text formatters which you may wish to use.

**unset**

unset The unset command removes the definitions of shell variables.

**variable expansion**

See variables and expansion.

**variables**

Variables in **csh** hold one or more strings as value. The most common use of variables is in controlling the behavior of the shell. See path, noclobber, and ignoreeof for examples. Variables such as argv are also used in writing shell programs (shell command scripts).

**verbose**

The verbose shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The verbose variable is set by the shell's –v command line option.

**wc**

The wc program calculates the number of characters, words, and lines in the files whose names are given as arguments.

**while**

The while built-in control construct is used in shell command scripts.

**word**

A sequence of characters which forms an argument to a command is called a word. Many characters which are neither letters, digits, '–', '.' nor '/' form words all by themselves even if they are not surrounded by blanks. Any sequence of characters may be made into a word by surrounding it with '' characters except for the characters '' and '!' which require special treatment. This process of placing special characters in words without their special meaning is called quoting.

**working directory**

At any given time you are in one particular directory, called your working directory. This directory's name is printed by the **pwd** command and the files listed by **ls** are the ones in this directory. You can change working directories using chdir.

**write**

The write command is used to communicate with other users who are logged in to UNIX.

# Communicating With UNIX System Users

The UNIX system offers a choice of commands that enable you to communicate with other UNIX system users. Specifically, they allow you to send and receive messages from other users (on either your system or another UNIX system), exchange files, and form networks with other UNIX 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**, **uname**, and **uuname** |
| For exchanging files: | **uucp**, **uuto**, **uupick**, and **uustat** |
| For networking: | **ct**, **cu**, and **uux** |

# Exchanging Messages

To send messages, use the **mail** command. This command delivers 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 the **mail** command to read your message and reply at any time.

You can also use **mail** to send short files containing memos, reports, and so on. 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.)

# mail

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*<**return**>

where *login* is the recipient's login name on a UNIX system. This login name can be either of the following:

- a login name if the recipient is on your system (for example, **bob**)
- a system name and login name if the recipient is on another UNIX system that can communicate with yours (for example, **sys2!bob**)

For the moment, assume that the recipient is on the local system. (We will deal with sending mail to users on remote systems later.) Type the **mail** command at the system prompt, press <**return**>, and start typing the text of your message on the next line. When you have finished typing it, send the message by typing a period (.) or a <**ctrl-d**> at the beginning of a new line.

The following example shows how this procedure will appear on your screen.

```
% mail phyllis<return>
My meeting with Smith's<return>
group tomorrow has been moved<return>
up to 3:00 so I won't be able to<return>
see you then. Could we meet<return>
in the morning instead?<return>
.<return>
%
```

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 recipient's login, the **mail** command will not be able to deliver your mail. Instead, it will print two messages telling you that it has failed and that it is returning your mail. Then it will return 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, say 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<return>
The meeting has been changed to 2:00.
.<return>
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<return>
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 Mail" later in this chapter.

## Sending Mail to One Person

The following screen shows a typical message.

```
% mail tommy<return>
Tom,<return>
There's a meeting of the review committee<return>
at 3:00 this afternoon. D.F. wants your<return>
comments and an idea of how long you think<return>
the project will take to complete.<return>
B.K.<return>
.<return>
%
```

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 Mail" in this chapter.

You can practice using the **mail** command by sending mail to yourself. Type in the **mail** command and your login ID, and then write a short message to yourself. When you type the final period or **<ctrl-d>**, the mail will be 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<return>
Call Accounting and find out<return>
why they haven't returned my 1985 figures!<return>
.<return>
%
```

When you log in the next day, a notice will appear 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<return>
Diamond cutters,<return>
The game is on for tonight at diamond three.<return>
Don't forget your gloves!<return>
Your Manager<return>
.<return>
%
```

Figure 9-1 summarizes the syntax and capabilities of the **mail** command.

| **Command Recap**<br>**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 **<ctrl-d>** (followed by **<return>**) at the beginning of a new line sends the message. | |

Figure 9-1: Summary of Sending Messages with the **mail** Command

## Sending Mail to Remote Systems

Until now we have assumed that you are sending messages to users on the local UNIX 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 simply by adding the name of the recipient's system before the login ID on the command line.

**mail sys2!bob<return>**

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:

- the name of the remote system
- whether or not your system and the remote system communicate
- the recipient's 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 recipient's 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<return>**

The command will respond with the name of the system. For example:

```
% uname –n<return>
dumbo
%
```

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<return>**

This generates a list of remote systems with which your system can communicate. If the recipient's 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***<return>**

(Here *system* is the recipient's system name.) If **grep** finds the specified system name, it prints it on the screen. For example:

```
% uuname | grep dumbo<return>
dumbo
%
```

This means that **dumbo** can communicate with your system. If **dumbo** does not communicate with your system, **uuname** returns a prompt.

```
% uuname | grep dumbo<return>
%
```

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 dumbo. Verify that dumbo can communicate with your system and send your message. The following screen shows both steps.

```
% uuname | grep dumbo<return>
dumbo
% mail dumbo!sarah<return>
Sarah,<return>
The final counts for the writing seminar<return>
are as follows:<return>
<return>
Our department – 18<return>
Your department – 20<return>
<return>
Tom<return>
.<return>
%
```

Figures 9-2 and 9-3 summarize the syntax and capabilities of the **uname** and **uuname** commands, respectively.

| Command Recap<br>**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. | |

Figure 9-2: Summary of the **uname** Command

* See the **uname**(1) manual page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

| Command Recap<br>**uuname** – displays a list of networked systems | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **uuname** | none | none |
| **Description:** | **uuname** displays a list of remote systems that can communicate with your system. | |

Figure 9-3: Summary of the **uuname** Command

# Managing Incoming Mail

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
```

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<return>**

The messages will be displayed 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 **<ctrl-s>**. This will freeze the screen, giving you a chance to read. When you are ready to continue, type **<ctrl-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 **<return>** after the question mark.

```
?<return>
```

The current message remains in your mailbox and the next message is displayed. 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<return>
```

The message is deleted from your mailbox. If there is another message waiting, it is then displayed. 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<return>
```

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<return>
```

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 pathname. For example:

```
? s project1/memo<return>
```

This is a relative pathname 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 pathnames when saving mail messages. (For instructions on using pathnames, see Chapter 3.)

To quit reading messages, enter the response shown after the question mark:

? **q<return>**

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 **<break>**. The **mail** command will stop the display, print a ? prompt, and wait for a response from you.

Figure 9-4 summarizes the syntax and capabilities of the **mail** command for reading messages.

| **Command Recap**<br>**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 (the 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 *IRIS-4D User's Reference Manual.* | |

Figure 9-4: Summary of Reading Messages with the **mail** Command

* See the **mail**(1) manual page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

# 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 for transferring a 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: the 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***<return>**

(For further information on input redirection, see Chapter 7.) Here *login* is the recipient's 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<return>
%
```

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<return>
%
```

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 user's login name. Separate the system name and the login name with an ! (exclamation point):

**mail** *system*!*login* **<** *filename***<return>**

For example:

```
% mail dumbo!wombat < agenda<return>
%
```

The system prompt on the second line means that your message (containing the file) has been queued for sending.

## 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 will 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 user's 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 pathnames 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 recipient's 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.

## Getting Ready: Do You Have Permission?

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 file's 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<return>
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<return>
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<return>
```

Now check the permissions again with the **ls -l** command:

```
% ls -l chicken<return>
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.

## The uucp Command

The command **uucp** (short for UNIX-to-UNIX system copy) 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), and 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 or
- multiple files to a directory

To deliver your file(s), **uucp** must know the full pathname of both the *source-file* and the *destination-file*. However, this does not mean you must type out the full pathname 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** will expand them to full pathnames.

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. (We'll 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 pathname.

How do you specify the *destination-file*? Because it is on a remote system, the *destination-file* must always be specified with a pathname 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, let's say 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 pathname, **chap1**. Specify the *destination-file* in any of the ways shown in the following command lines:

- Specify the *destination-file* with its full pathname:

  **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 recipient's home directory) but without a name for the new file; **uucp** will give 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 recipient's subdirectory in the public directory on the remote system.

  **uucp chap1 minnie! /wsm**

  (The file will go to **minnie!/usr/usr/spool/uucppublic/wsm**)

## Sample Usage of Options with 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<return>
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 pathname **/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 will get this message: `uucp failed completely`.

The job ID (`eagleN3f45`) is displayed 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 "How the **uucp** Command Works" 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 pathname 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 pathname. For example:

  **eagle!~/gws/minutes**

  is interpreted as:

  **/usr/spool/uucppublic/gws/minutes**

## How the uucp Command Works

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 the command's 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 *IRIS-4D System Administrator's Guide* and the *IRIS-4D 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 for transferring the file(s). The data file is simply 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 then transfers the file(s) that you are sending to the remote computer; the remote **uucico** places the file in the specified pathname(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.

Figure 9-5 summarizes the syntax and capabilities of the **uucp** command.

| Command Recap<br>**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 of representing pathnames as arguments, it is recommended that you type full pathnames to avoid confusion. | |

Figure 9-5: Summary of the **uucp** Command

* See the **uucp**(1) manual page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

## The 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*<**return**>

where *filename* is the name of the file to be sent, *system* is the recipient's system, and *login* is the recipient's 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*<**return**>

## Sending a File: the m Option and uustat Command

Now that you know how to determine if a file is transferable, let's take an example and see how the whole thing 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 job's 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** | recipient's login name |
| **sys2** | recipient's 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<return>
%
```

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<return>
%
```

This option sends a **mail** message back to you when the job has reached the recipient's system. The message may look something like this:

```
% mail<return>
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<return>
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 recipient's 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***<return>**

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<return>
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 recipient's system. Other status messages and options for the **uustat** command are described in the *IRIS-4D User's Reference Manual.*

That is all there is to sending files. To practice, try sending a file to yourself.

Figures 9-6 and 9-7 summarize the syntax and capabilities of the **uuto** and **uustat** commands, respectively.

| Command Recap<br>**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 file's parent directory must have read and execute permissions for others.<br><br>The **–m** option notifies the sender by mail when the file has arrived at its destination. | |

Figure 9-6: Summary of the **uuto** Command

* See the **uuto**(1) manual page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

| Command Recap<br>**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. | |

Figure 9-7: Summary of the **uustat** Command

* See the **uustat**(1) manual page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

## Receiving Files Sent with uuto

When a file sent by **uuto** reaches the public directory on your UNIX 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 system's 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 pathname 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<return>**

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, say the owner of login **marie** issues the **uupick** command to retrieve the **money** file. The command will respond as follows:

```
% uupick<return>
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<return>
%
```

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<return>
```

If there are other files waiting to be moved, the next one is displayed, 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:

```
?
<return>
```

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<return>
```

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<return>
```

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 *IRIS-4D User's Reference Manual.*

Figure 9-8 summarizes the syntax and capabilities of the **uupick** command.

| Command Recap<br>**uupick** – searches for files sent by **uuto** or **uucp** | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **uupick** | –s | 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 *IRIS-4D User's Reference Manual.* | |

Figure 9-8: Summary of the **uupick** Command

# 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 such as 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 UNIX system.

## Connecting a Remote Terminal

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, that is, 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 will hang 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, follow this format:

**ct** [*options*] *telno*<**return**>

The argument *telno* is the telephone number of the remote terminal.

## Sample Command Usage

Suppose 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<return>**

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** will call the modem, using a dialer operating at a speed of 1200 baud. If a dialer is not available, the **–w5** option will cause **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 imagine that 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<return>**

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.

Figure 9-9 summarizes the syntax and capabilities of the **ct** command.

| **Command Recap**<br>**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. | |

Figure 9-9: Summary of the **ct** Command

* See the **ct**(1) manual page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

## Calling Another UNIX System

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 on the command line. 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 will still be 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 will 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 *IRIS 4D User's Reference Manual* for details.)

## Command Line Format

To execute the **cu** command, follow this format:

**cu** [*options*] *telno* / *systemname***<return>**

The components of the command line are:

*telno* the telephone number of a remote computer

Equal signs (=) represent secondary dial tones and dashes (–) repreent four-second delays.

*systemname* 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 together 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. Figures 9-10 and 9-11 show the commands you can execute while connected to a remote computer through **cu**.

| String | Interpretation |
|---|---|
| ~. | Terminate the link. |
| ~! | Escape to the local computer without dropping the link. To return to the remote computer, type **<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 pathname 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**). |

Figure 9-10: Command Strings for Use with **cu** (Sheet 1 of 2)

| 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). |

Figure 9-11: Command Strings for Use with **cu** (Sheet 2 of 2)

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 be 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

Suppose you want to connect your computer to a remote computer called **eagle**. The phone number for eagle is 847–7867. Enter the following command line:

**cu –s1200 9=8477867<return>**

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. Suppose 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 will also be called **proposal**.

**~ $take proposal<return>**

The **put** command allows you to do the opposite: copy files from the local computer to the remote computer. Say 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<return>**

In this case, 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 will be called **minutes.9-18**.

Figure 9-12 summarizes the syntax and capabilities of the **cu** command.

| Command Recap<br>**cu** – connects computer to remote computer | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **cu** | –s and others* | *telno* (or) *systemname* |
| **Description:** | **cu** connects your 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. | |

Figure 9-12: Summary of the **cu** Command

* See the **cu**(1) manual page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

# Executing Commands on a Remote System

The command **uux** (short for UNIX-to-UNIX system command execution) allows you to execute UNIX 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, follow this format:

**uux** [*options*] *command–string*<**return**>

The *command–string* is made up of one or more arguments. All special shell characters (such as "<>|^") 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. Arguments without a systemname are read as command arguments. A file name may be either a full pathname 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 computer by entering:

**pr minutes | uux –p host!lp<return>**

This command line queues the file **minutes** to be printed on the area printer of the computer **host.**

Figure 9-13 summarizes the syntax and capabilities of the **uux** command.

| **Command Recap**<br>**uux** – executes commands on a remote computer | | |
|---|---|---|
| *command* | *options* | *arguments* |
| **uux** | **–l, –p**, and others* | *command-string* |
| **Description:** | **uux** allows you to run UNIX 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** command. Check with your system administrator to find out if users on your system have been granted permission to run other commands. | |

Figure 9-13: Summary of the **uux** Command

* See the **uux**(1) manual page in the *IRIS-4D User's Reference Manual* for all available options and an explanation of their capabilities.

# The UNIX System Files

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 UNIX system files are organized in a hierarchy; their 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.

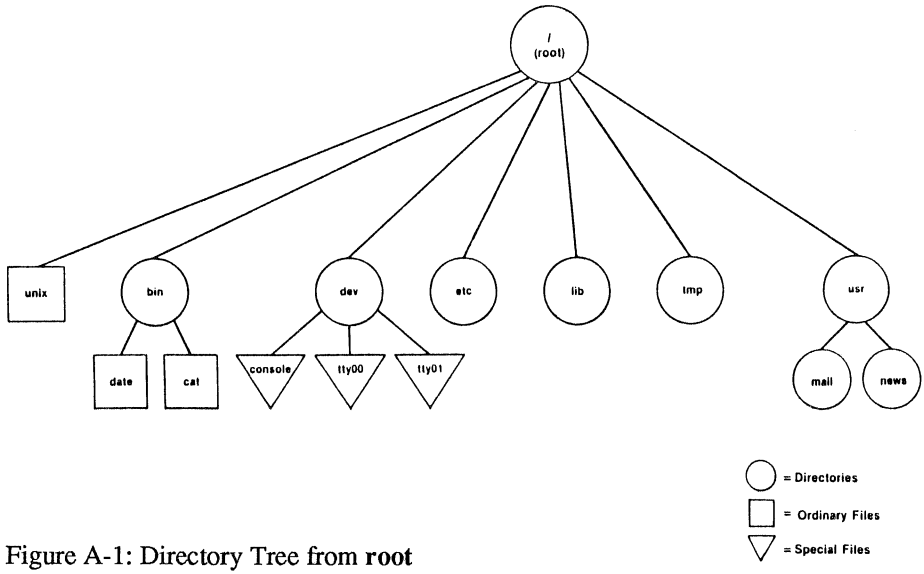



Figure A-1: Directory Tree from **root**

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 UNIX system installations and are provided and maintained by the operating system.

In addition to this standard set of directories, your UNIX system may have other system directories. To obtain a list of the directories and files in the **root** directory on your UNIX system, type the following command line:

**ls –l /<return>**

To move around in the file structure, you can use pathnames. For example, you can move to the directory **/bin** (which contains UNIX system executable files) by typing the following command line:

**cd /bin<return>**

To list the contents of a directory, issue one of the following command lines:

| | |
|---|---|
| **ls<return>** | for a list of file and directory names |
| **ls –l<return>** | 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<return>** | for a short listing |
| **ls –l /bin<return>** | 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.

# UNIX System Directories

**/** The source of the file system (called **root** directory)

**/bin** Contains many executable programs and utilities, such as the following:

**cat**
**date**
**login**
**grep**
**mkdir**
**who**

**/lib** Contains available program libraries and language libraries, such as

| | |
|---|---|
| **libc.a** | system calls, standard I/O |
| **libm.a** | math routines and support for languages such as C, Fortran, and Basic. |

**/dev** Contains special files that represent peripheral devices, such as:

| | |
|---|---|
| **console** | console |
| **lp** | line printer |
| **tty*n*** | user terminal(s) |
| **dsk/*** | disks |

**/etc** Contains programs and data files for system administration

**/tmp** Contains temporary files, such as the buffers created for editing a file

**/usr** Contains the following subdirectories which, in turn, contain the data listed below:

| | |
|---|---|
| **news** | important news items |
| **mail** | electronic mail |
| **spool** | files waiting to be printed on the line printer |

# Basic UNIX System Commands

**at** Request that a command be run 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.

A sample format is:

**at 8:45am Jun 09<return>**
*command1* **<return>**
*command2* **<return>**
**<^d>**

If you use the **at** command
without the date,
the command executes within twenty-four 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. A sample format of this command is:

**batch<return>**
*command1* **<return>**
*command2* **<return>**
**<ctrl-d>**

You can use a shell script for a command in **batch**(1).
This may be useful and save time 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 **<ctrl-s>**; type **<ctrl-q>** to restart the output. To interrupt the output and return to the shell on an ASCII terminal, press **<break>** or **<delete>**.

**cd** Change directory from the current one to your home directory. If you include a directory name, the directory will change from the current one to the directory specified. By using a pathname 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 the terminal, including the carriage return, and return 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 pattern and print 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 *IRIS-4D User's Reference Manual* for details.

**lp** Print out the contents of a specified file on a line printer.

**lpstat** Display the status of any requests made to the line printer. Options are available for requesting detailed information.

**ls** List the names of all files and directories except those whose names begin with a dot ( **.** ). Options are available for listing detailed information about the files in the directory. (See the **ls**(1) entry in the *IRIS-4D 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 such as 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 **<ctrl-d>** to end the message and send it to the

recipient. Press **<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 *IRIS-4D 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 will continue executing after you log off the system. Error messages are placed in a file called **nohup.out**.

**pg** Display the contents of a specified file on your terminal, a 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 pathname 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 will 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 *IRIS-4D User's Reference Manual* for the ability to remove directories that are not empty.)

**sort** Sort a file in ASCII order and display the results on your terminal. ASCII order is as follows:

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 *IRIS-4D 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 (with suffixes, prefixes, and so on) are displayed.

**uname** Display the name of the UNIX system on which you are currently working.

**uucp** Send a specified file to another UNIX system. (See the **uucp**(1) page in the *IRIS-4D User's Reference Manual* for details.)

**uuname** List the names of remote UNIX systems that can communicate with your UNIX 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 displays a **?** prompt.

**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 UNIX 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 *IRIS-4D User's Reference Manual* for details.

# Summary of Shell Command Language

This appendix is a summary of the shell command language and programming constructs discussed in Chapter 7, "The Bourne Shell Tutorial" and Chapter 8, "An Introduction to the C Shell." The first section reviews metacharacters, special characters, input and output redirection, variables, and processes. 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 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 turns off the special meaning of all characters. |
| "..." | Enclosing double quotes turns 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 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. **<ctrl-d>** ends the **batch** command.

**at** Submits the following commands to be executed at a specified time. **<ctrl-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.

## 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* **$HOME/bin/***filename*
Moves your file to the **bin** directory in your home directory. This **bin** holds executable shell programs that you want to be accessible. Make sure the **PATH** variable in your **.profile** file specifies this **bin**. If it does, the shell will search in **$HOME/bin** for your file when you try to execute it. If your **PATH** variable does not include your **bin**, the shell will not know where to find your file and your attempt to execute it will fail.

**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 assigned by the shell from the command line arguments invoking the program.

**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<return>
        in this list of values<return>
do the following commands<return>
        command 1<return>
        command 2<return>
            .<return>
            .<return>
        last command<return>
done<return>
```

## While Loop

```
while command list<return>
do<return>
    command1<return>
    command2<return>
        .<return>
        .<return>
    last command<return>
done<return>
```

## If...Then

```
if this command is successful<return>
then command1<return>
    command2<return>
        .<return>
        .<return>
    last command<return>
fi<return>
```

## If...Then...Else

```
if command list<return>
    then command list<return>
    else command list<return>
fi<return>
```

## Case Construction

```
case word<return>
in<return>
    pattern1)<return>
        command line 1<return>
            .<return>
            .<return>
        last command line<return>
    ;;<return>
    pattern2)<return>
        command line 1<return>
            .<return>
            .<return>
        last command line<return>
    ;;<return>
    pattern3)<return>
        command line 1<return>
            .<return>
            .<return>
        last command line<return>
    ;;<return>
esac<return>
```

## 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.

# Setting the TERM Variable

Because some commands are terminal dependent, the UNIX 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 have put the name of your terminal into this variable, the system will be able to execute all programs in a way that is suitable for your terminal.

This method of telling the UNIX 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<return>
% export TERM<return>
% tput init<return>
```

These lines must be executed in the order shown and the procedure must be repeated every time you log in. To expedite this process most users put these lines into a file called **.profile** 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 UNIX system shell what type of terminal you are using. The **tput init** command line instructs your terminal to behave in ways the UNIX system expects a terminal of that type to behave. For example, it sets the terminal's left margin and tabs, if those capabilities exist for the terminal.

The **tput** command uses the entry in this database 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 *IRIS-4D Programmer's Reference Manual.*

## Acceptable Terminal Names

The UNIX 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 within this range.

You must also verify that the name you put into the **TERM** variable is a recognized terminal name. 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<return>**

If your system supports your terminal it will respond with the complete name of your terminal. Otherwise, you will get an error message.

To find an acceptable name that you can put in the **TERM** variable, find a listing for your terminal in either of two directories: **/usr/lib/terminfo** or **/usr/lib/.COREterm**. Each of these directories is a collection of files with single-character names. Each file, in turn, holds a list of terminal names that all begin with the name of the file. Find the file whose name matches the first character of your terminal's name. Then list the file's contents and look for your terminal.

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

Suppose your terminal is an AT&T 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/.COREterm/A** directory. The following screen shows which commands you need to do this:

```
% tput –T5425 longname<return>
AT&T 4425/5425
% cd /usr/lib/.COREterm/A<return>
% 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<return>
% export TERM<return>
% tput init<return>
%
```

The UNIX system now knows what type of terminal you are using and will execute commands appropriately.

# 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 UNIX 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 (such 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 UNIX 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 (**<return>**). 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 control key and typing a character. Control characters are often used for special purposes. For instance, when viewing a long file on your screen with the **cat** command, typing ctrl-s stops the display so you can read it, and typing ctrl-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 UNIX 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.)

**electronic mail**
The feature of an operating system that allows computer users to exchange written messages via the computer. The UNIX system **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 UNIX system. Your environment includes those things that personalize your login and allow you to interact in specific ways with the UNIX 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 UNIX system 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 UNIX system files by name. See **ordinary file**, **permanent file**, and **executable file**.

**file name**
A sequence of characters that denotes a file. (In the UNIX 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 UNIX 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 UNIX system uses the full-duplex setting.

**full pathname**
A pathname that originates at the root directory of the UNIX system and leads to a specific file or directory. Each file and directory in the UNIX system has a unique full pathname, sometimes called an absolute pathname. See **pathname**.

**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 <ctrl-d> are examples.

**home directory**
The directory in which you are located when you log in to the UNIX 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 UNIX system) that can handle immediate-response communication between you and the computer. In other words, you interact with the computer from moment to moment.

**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 UNIX 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 UNIX 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 (for example, 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.

**multiuser**
The ability of an operating system to support several users on the system at the same time.

**nroff**
A text formatter available as an add-on to the UNIX system. You can use the **nroff** program to produce a formatted on-line copy or a printed copy of a file. See **text formatter.**

**operating system**
The software system on a computer under which all other software runs. The UNIX 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 UNIX 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 executed. 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.

**pathname**
A sequence of directory names separated by the slash character (/) and ending with the name of a file or directory. The pathname 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 make use of 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. It is named for the character | that redirects the output. 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 UNIX 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 UNIX system default prompt is the percent sign character (%).

**printer**
An output device that prints the data it receives from the computer on paper.

**process**
Generally a program that is at a stage of execution. In the UNIX system, it also refers to the execution of a computer environment, including contents of memory, 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 UNIX system to read and interpret your input while sending output information to your terminal in response to previous input. The UNIX system separates input from output and processes each correctly.

**relative pathname**
The pathname 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 UNIX system default secondary prompt is the "greater than" character (>).

**shell**
A UNIX 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 such as 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 will then come from the specified file.

**standard output**
An open file that is normally connected directly to a primary output device, such as 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 will then go to the specified file.

**string**
Designation for a particular group or pattern of characters, such as 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 UNIX system runs; sometimes referred to as a super-user.

**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 UNIX 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. These special commands tell the formatter to justify margins, start new paragraphs, set up lists and tables, place figures, and so on. Two text formatters available as add-ons to your UNIX system are **nroff** and **troff.**

**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 available as an add-on to the UNIX system. The **troff** program drives a phototypesetter to produce high-quality printed text from a file. See **text formatter**.

**tty**
Historically, the abbreviation for a teletype terminal. Today, it is generally used to denote a user terminal.

**user-defined**
Something determined by the user.

**user-defined variable**
A named variable given a value by the user. See **variable**.

**UNIX system**
A general-purpose, multiuser, interactive, time-sharing operating system developed by AT&T Bell Laboratories. The UNIX system allows limited computer resources to be shared by several users and efficiently organizes the user's interface to a computer system.

**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**.

**Silicon Graphics, Inc.**

Date _____

Your name _____

Title _____

Department _____

Company _____

Address _____

_____

_____

Phone _____

**COMMENTS**

Manual title and version _____

_____

Please list any errors, inaccuracies, or omissions you have found in this manual

_____

_____

_____

_____

Please list any suggestions you may have for improving this manual

_____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 45 MOUNTAIN VIEW, CA

POSTAGE WILL BE PAID BY ADDRESSEE

**Silicon Graphics, Inc.**
Attention: Technical Publications
2011 N. Shoreline Boulevard
Mountain View, CA 94039-7311