User's Guide

CONCURRENT COMPUTER CORPORATION™

0890428-010

February 1997

Revision History:		Level:	Effective With:
Original Release	-- July 28, 1994	000	PowerUX 1.1
Reissue 1	-- February 1997	010	PowerMAX OS 4.1

Contents

Tables

1
Introduction

1
Introduction

Introduction

This introduction describes each of the chapters and appendixes in this guide, and describes the notation conventions used throughout.

Contents of this Guide

This guide contains three major parts:

- overview of the UNIX operating system
- tutorials on the main tools available on the UNIX® system
- reference section.

This guide also contains the security-related information that comprises the *Secure Facility User's Guide*.

Secure Facility User's Guide

The *Secure Facility User's Guide* meets the standards for this type of document described in the *Trusted Computer System Evaluation Criteria* (DoD 5200.28.STD, 1985). It shows the intended use of the features available to all users and describes the overall design of the system.

To effectively use the security features of the system, you must read at least the following sections of this guide:

- "What is the UNIX System?"
- "Basics for UNIX System Users"
- "Using the File System"
- "Managing Files Securely."

Chapter 8, "LP Print Service Tutorial" and Chapter 10, *"Electronic Mail Tutorial"* may also be useful.

System Overview

The three chapters of the *User's Guide* 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.

- What Is the UNIX System? provides an overview of the operating system
- Basics for UNIX System Users discusses the general rules and guidelines for using the UNIX system, including:
  - using your terminal
  - obtaining a system account
  - establishing contact with the UNIX system
- Using the File System gives a working perspective of the file system, including:
  - commands for building your own directory structure
  - accessing and manipulating the subdirectories and files
  - examining the contents of other directories in the system.

UNIX System Tutorials

The second part of the *User's Guide* contains tutorials. The tutorials assume you understand the concepts introduced in the beginning chapters. They also provide hands-on exercises designed to give you a more thorough understanding of the information.

- "Overview of the Tutorials" highlights UNIX system capabilities including:
  - command execution
  - text editing
  - electronic communication.
- "*Line Editor (ed) Tutorial*" shows how to use the `ed` text editor to create and modify text on a video display terminal or paper printing terminal
- "*Screen Editor (vi) Tutorial*" shows how to use the visual text editor to create and modify text on a video display terminal.

NOTE

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

- *Chapter 8, "LP Print Service Tutorial"* shows how to print hard copies of files using the LP print service and provides instructions for other print services including:
  - enabling and disabling a printer
  - modifying the format mode of printed pages
  - controlling the printing process
- Chapter 9, "P*rogramming with the UNIX System Shell*" shows how to use the shell as a command interpreter.
- Chapter 10, *"Electronic Mail Tutorial"* shows how to use electronic mail and its associated commands.
- Chapter 11, "*Remote Services Tutorial*" shows how to perform operations on a remote host.
- Chapter 12, "*Communication Tutorial*" shows how to send messages and files to users of both your UNIX system and other UNIX systems
- Chapter 13, "*Programming with awk*" describes a programming language that enables you to handle easily the tasks associated with data processing and information retrieval
- Chapter 14,"*Managing Files Securely*" describes security levels and how to manage your work efficiently and securely in a multi-level environment.

Reference Information

Several appendixes and a glossary of UNIX system terms are also provided for reference:

- Appendix A, "*Summary of the File System*," shows how information is stored in the UNIX operating system.
- Appendix B, "S*ummary of UNIX System Commands*," describes, in alphabetical order, each UNIX system command discussed in the *User's Guide.*
- Appendix C, "*Quick Reference to ed Commands*," summarizes commands for the line editor, **`ed.`**
- Appendix D, "*Quick Reference to vi Commands*," summarizes commands for the full screen editor, **`vi`**
- Appendix E, "*Summary of Shell Command Language*," summarizes the shell command language, and notation.
- Appendix F, "*Setting Up the Terminal*," explains how to configure your terminal for use with the UNIX system, and create multiple windows on the screens of terminals with windowing capability.
- The Glossary provides definitions of the UNIX system terminology used in this book.

Syntax Notation

The following notation is used throughout this guide:

- User input (commands, options and arguments for commands, environment variable names, and directory and file names files) and UNIX system output (prompt signs and responses to commands) appear in a `constant width font`.
- Names of variables to which values must be assigned (*password*) and names of books appear in *italic.*
- Keyboard keys and ASCII graphic input are shown in a key format, for example, <`RETURN`>, <`TAB`>, and <!>
- Control characters are shown as a key because they do not appear on the screen when typed. The control key, which appears on many keyboards, is represented by the string <`CTRL`>. 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 typing a lowercase d; the letter d will not appear on the screen.
- Command options and arguments that are optional, such as `[`**`-msCj`**], are enclosed in brackets.
- The vertical bar (|) separates optional arguments, from which you may choose one. For example, when a command line has the following format:

  command `[`*arg1* `|` *arg2*`]`

  You may use either *arg1* or *arg2* when you issue the *command.*
- Ellipses (`...`) after an argument mean that more than one argument may be used on a single command line.
- Arrows (↑) on the screen (shown in examples in the "Screen Editor (vi) Tutorial" chapter, represent the cursor.
- A parenthetical number immediately after the name of a command refers to the part of a UNIX system reference manual where the command is documented in the manual pages. There are three online reference manuals:
  - *Command Reference*
  - *Operating System API Reference*
  - *System Files and Devices*

  For example, the notation **`cat(1)`** refers to the manual page in Section 1 of the online *Command Reference* that documents the **`cat`** command. Online manual pages (man pages) can be viewed using the **`man`** command. For example, entering "**`man cat`**" will display the **`cat`** man page on your screen.

Although it may not be the actual prompt for all systems, in sample commands, the `$` sign is used as the shell command prompt. You are not meant to type prompts; they are produced by the system. (The $ sign is also used to reference the value of positional parameters and named variables.)

In all chapters, full and partial screens are used to display examples of how your terminal screen will look when you interact with the UNIX system. The input (characters typed by you) and output (characters printed by the UNIX system) are shown in these screens using the conventions listed above.

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

Referenced Documentation

The following manuals are referenced in this manual:

PowerMAX OS Programming Guide	0890423
Device Driver Programming	0890425
System Administration Volume 1	0890429
System Administration Volume 2	0890430
Network Administration	0890432
Compilation Systems Volume 1 (Tools)	0890459
Compilation Systems Volume 2 (Concepts)	0890460
Trusted Computer System Evaluation Criteria	DoD 5200.28.STD, 1985

2
What Is the UNIX System?

2
What Is the UNIX System?

Overview of the UNIX Operating System

The UNIX operating system is a set of programs (or software) that controls the computer, acts as the link between you and the computer and provides tools to help you do your work. Designed to provide an uncomplicated, efficient, and flexible computing environment, the UNIX system offers several advantages:

- a general purpose system that performs a wide variety of jobs or applications
- an interactive environment that allows you to communicate directly with the computer and receive immediate responses to your requests and messages
- a multi-user environment that allows you to share the resources of the computer with other users without sacrificing productivity

NOTE

> This technique is called timesharing. The UNIX system interacts with 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 at the same time.

The UNIX system has four major components:

kernel	The kernel is a program that makes up the core of the operating system; it coordinates the internal functions of the computer (such as allocating system resources). The kernel works invisibly; you are never aware of it while you are doing your work.
shell	The shell is a program that acts as a liaison between you and the kernel by interpreting and executing your commands. Because it reads your input and sends you messages, it is described as interactive.
commands	Commands are the names of programs that you request the computer to execute. Packages of programs are called tools. The system provides tools for jobs such as creating and changing text, writing programs, developing software tools, and exchanging information with others via the computer.

file system The file system is the collection of all the files available on your computer. It allows you to store and retrieve information easily.

Figure 2-1 is a model of the UNIX system. Each circle represents one of the main components: the kernel, the shell, and the commands (user programs). 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, the file system.

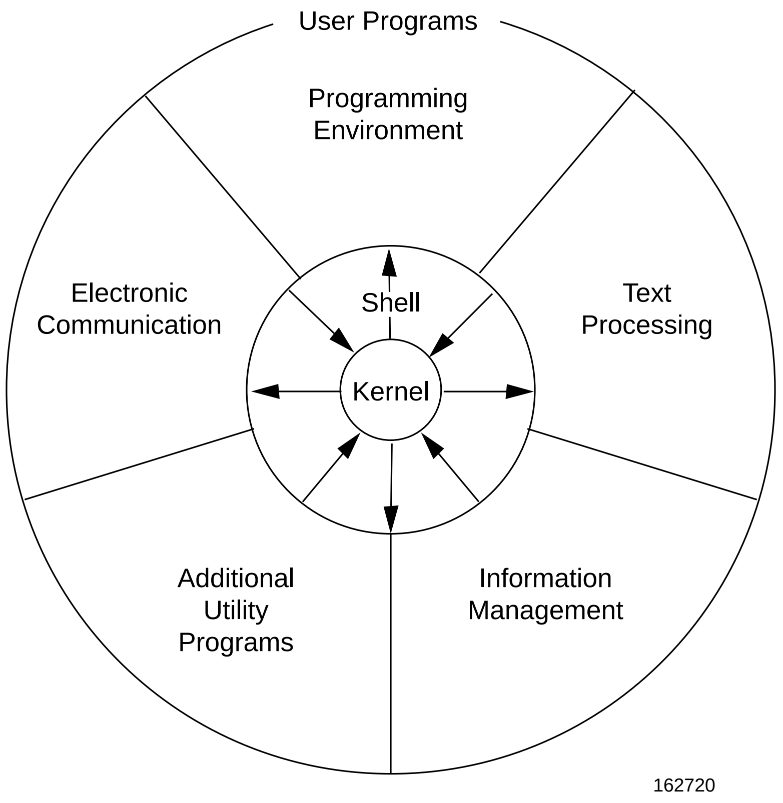


Figure 2-1. Model of the UNIX System

The Kernel

The nucleus of the UNIX system is called the kernel. The kernel

- controls access to the computer
- manages computer memory
- maintains the file system
- allocates computer resources among users.

Figure 2-2 shows a functional view of the kernel.

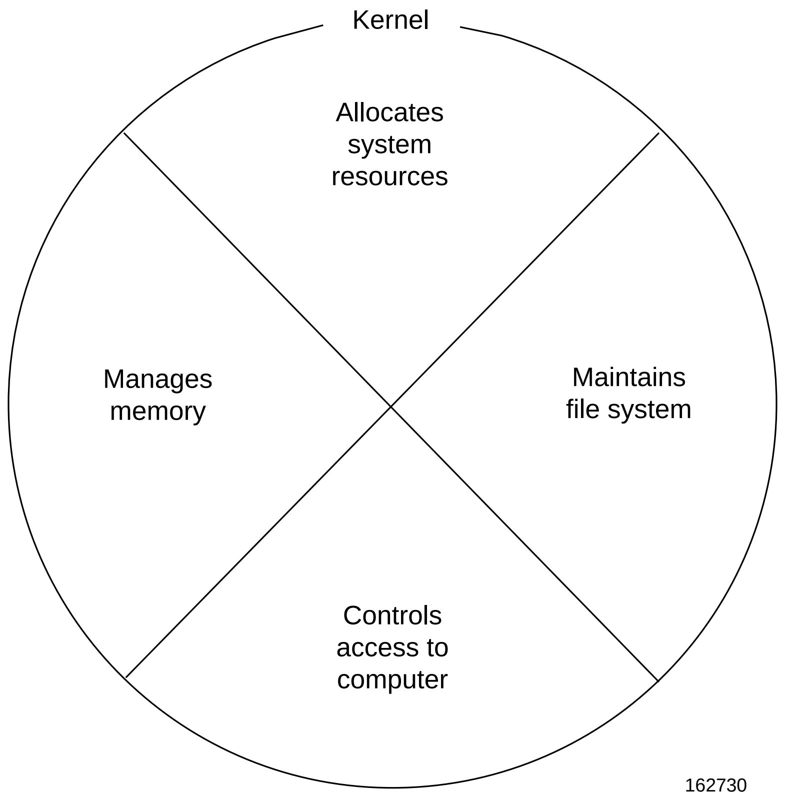


Figure 2-2. Functional View of the Kernel

UNIX System Commands

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

This chapter introduces many of the UNIX system programs and tools you will use regularly. If you need additional information on these or other standard programs, refer to the manual pages available in the online *Command Reference.*

For information on tools and routines related to programming and software development, refer to the manual pages in the online *Operating System API Reference.*

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

Command Functions

The outer circle of the UNIX system model in Figure 2-1 organizes the system programs and tools into functional categories:

programming environment	Several UNIX system programs establish a friendly programming environment by providing interfaces between the UNIX system and programming languages and by supplying utility programs.
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 suitable for publication.
information management	The system provides programs that allow you to create, organize, and remove files and directories.
additional utility programs	Other tools generate graphics and perform calculations.
electronic communication	Several programs, such as **`mail`**, enable you to transmit information to other users and to other UNIX systems.

Executing Commands

To make your requests comprehensible to the UNIX system, each command must be presented in the correct format, or command line syntax. The command line 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, you must 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. The following is an example of command line syntax:

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

You must type at least two components on every UNIX system command line

- command name of the program you want to run
- <RETURN> key.

A command line may also contain either options or arguments, or both:

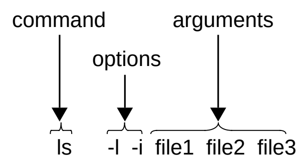
- an *option* modifies how the command runs
- an *argument* specifies data that the command processes, usually the name of a directory or file.

In command lines that include options and/or arguments, the component words are separated by at least one blank space. (Insert a blank by pressing the space bar.)

If an argument name contains a blank, enclose that name in 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, the **`ls`** command is used with two options, **`-l`** and **`-i`**, to list information about *file1, file2,* and *file3.*

The **`-l`** option displays information in a long format, including such things as mode, owner, and size. The **`-i`** option prints the inode number. (The UNIX system usually allows you to group options such as these to read **`-li`** if you prefer, and to enter them in any order.) In addition, three files (*file1, file2, and file3*) are specified as arguments. Although most options can be grouped together, arguments cannot.

See Table 2-1 for an example of proper sequence and spacing in command line syntax

Table 2-1. Command Line Syntax Sequence and Spacing

Incorrect	Correct
`ls`*file*	**`ls`** *file*
`ls-l` *file*	**`ls -l`** *file*
`ls -l i` *file*	**`ls -li`** *file*
	or
	`ls -l -i` *file*
`ls` *file1file2*	**`ls`** *file1 file2*

Regardless of the number of components, you must end every command line by pressing the <RETURN> key.

Figure 2-3 shows the flow of control when the UNIX system executes a command.

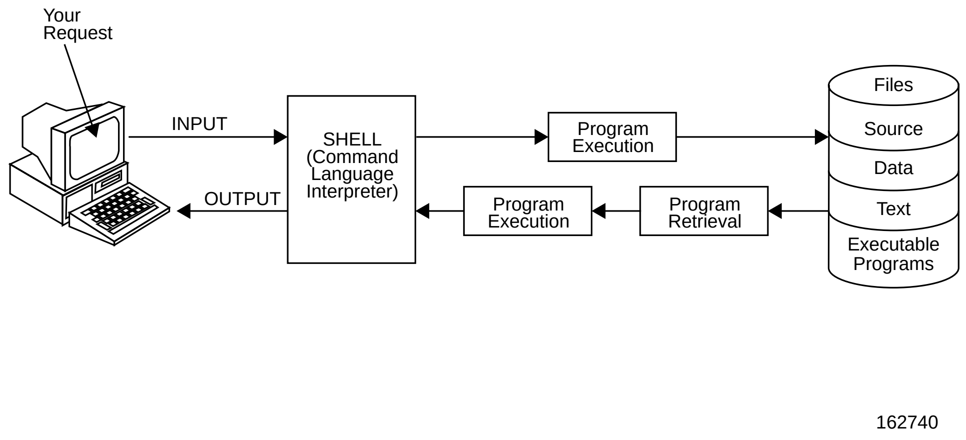


Figure 2-3. Execution of a UNIX System Command

To execute a command:

1. Enter a command line when a prompt (such as a `$` sign) appears on your screen.
2. The shell takes 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.
3. The kernel then follows the instructions in the program and executes the command you requested.
4. After the program has finished running, the shell signals that it is ready for your next command by printing another prompt.

The File System

The file system 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 Figure 2-4.

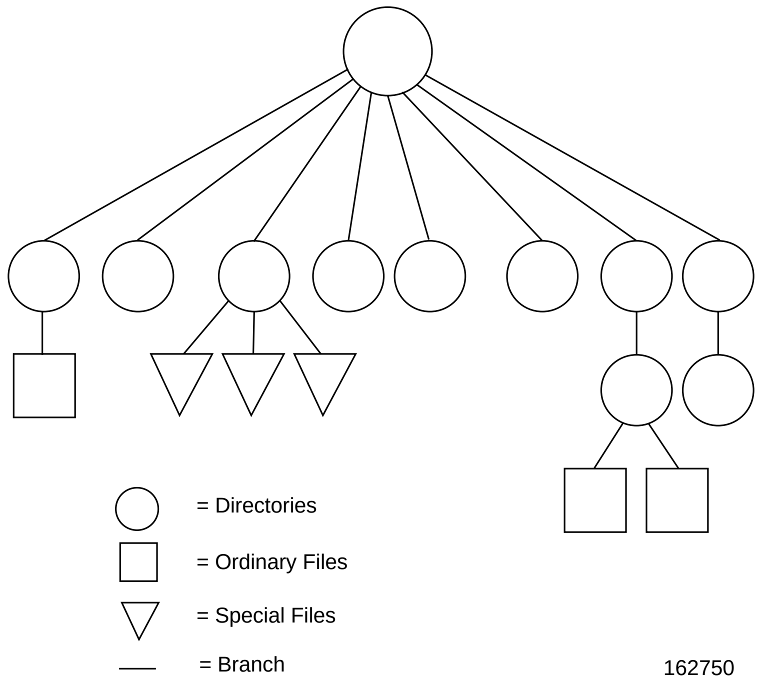


Figure 2-4. The Hierarchical Structure of the File System

The file, is the basic unit of the UNIX system. A file can be any of the following:

- ordinary file
- directory
- special file
- symbolic link.

(See Chapter 4, *Using the File System*.)

Ordinary Files

An ordinary file is a collection of characters that are treated as a unit by the system. Ordinary files are used to store any 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. After you have created a file, you can add to it, delete from it, or remove it entirely when you no longer need it.

Directories

A directory is a super-file that may contain files and other directories.

Usually the files it contains are related in some way. 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 your home directory; no one else except a privileged user can read or write files in it without your explicit permission, and you determine its structure.

The system also maintains several directories for its own use. The structure of these directories is much the same on all UNIX systems. These directories, which include 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 operating system file structure; all directories and files are arranged hierarchically under it.

Special Files

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

Symbolic Links

Symbolic links are files that point to other files. For more information about them and their uses, see the "*Symbolic Links*" chapter of the *Compilation System Manual.*

System Layout

To the UNIX system, all files are alike. It does not require you to define the type of file you have and to use it in a specified way or to consider how the files are stored (sequential, random-access, or binary files).

This makes the UNIX system file structure easy to use and simplifies your interaction with the system. For example, you need not specify memory requirements for your files, since the system automatically does this for you. 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.

Figure 2-5 shows an example of a typical file system. Notice that the root directory contains several important system directories.

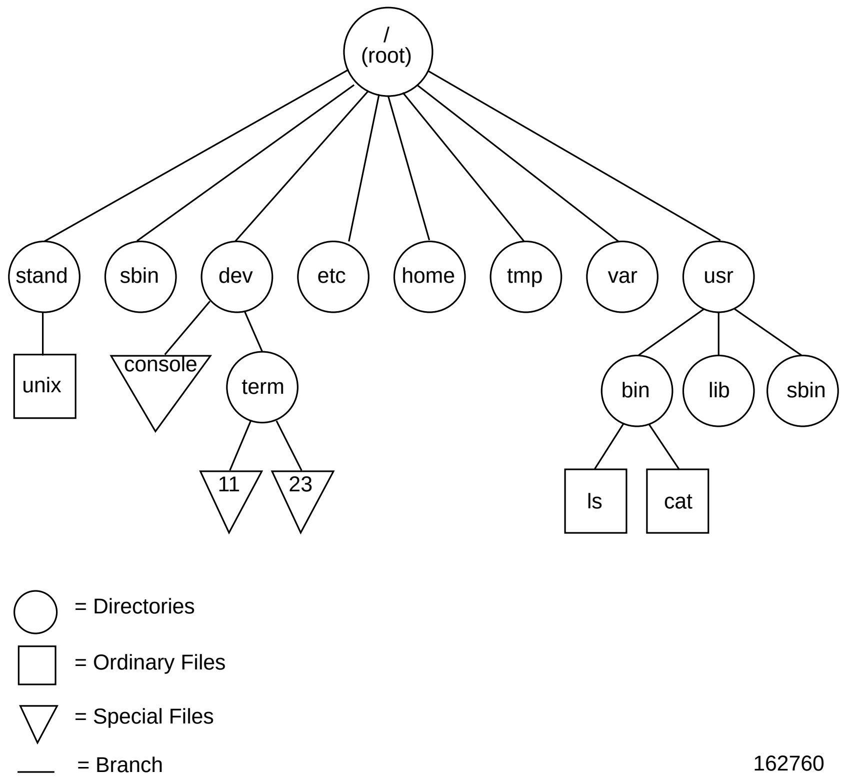


Figure 2-5. Example of a File System

`/stand`	contains bootable programs and data files used in the booting process
`/sbin`	contains essential executables used in the booting process and in manual system recovery
`/dev`	contains special files that represent peripheral devices such as the console, line printers, user terminals, and disks
`/etc`	contains machine-specific administrative configuration files and system administration databases
`/home`	the root of a subtree for user directories
`/tmp`	contains temporary files
`/var`	the root of a subtree for varying files such as log files
`/usr`	contains other directories, including `lib` and `bin`

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

Chapter 4, *Using the File System*, shows how to organize a file system directory structure, and access and manipulate files. Chapter 6, *Line Editor (ed) Tutorial*, and Chapter 7, *Visual Editor (vi) Tutoria*l, teach you how to create and edit files.

Security Concepts

A computer user is any person who interacts directly with a computer system. As a user, you should know how you are affected by security. You also need to understand the difference between basic security and enhanced security.

The UNIX system with the Enhanced Security Utilities installed includes special software that protects information on the system. This protective software is part of the Trusted Computing Base (TCB).

The TCB includes all software, firmware, and hardware that enforces security.

System Security Officer

Access to the TCB is determined by an administrator called the System Security Officer.

Users must contact the System Security Officer when they want to change their security level, privileges, or other TCB interactions.

Users can also contact the System Security Officer when they have difficulty interacting with the TCB.

Protecting Information

Security for a computing system means that the information on the system is protected from unauthorized disclosure or modification. If each user had a personal non-networked computing system that was kept locked up, each user's files would be secure. But isolation and physical security are not practical in most circumstances.

On a computer system that many people share, the simplest security mechanism would be to allow only the owner of a file to access that file. That would be inconvenient, however, because it would not take advantage of shared resource capabilities. For example, it would be wasteful for each user to have a private copy of each command. Commands are usually shared, but users often want to restrict access to the contents of files.

On a secure system, each user has a unique identity and a level of authorization associated with that identity. The computer system must have some way of identifying users, their level of authorization, and their files.

For the most part, while you are logged in, all data you enter, create, and process belongs to you. Data is stored in named files on the computer system. Each file you own is kept separate from the rest of your files and from the files belonging to other users.

A secure computer system must have a mechanism that decides access permissions based upon user identity and authorization.

There are many ways in which the security of a computer system can be violated. Unauthorized access to read or write files can be the result of

- abuse of privileges by administrators
- malicious programs (viruses) that gain privileges or access to files
- idle browsing of files that are inadequately protected.

Most computer systems provide some degree of basic security. However, the mechanisms supplied by the UNIX system Trusted Computing Base and the Enhanced Security Utilities provide specific, enhanced protection against these and other potential security hazards.

A review of basic security will provide a background for understanding the enhanced security available with the UNIX system.

Basic Security

A computer system enforces basic security by making access decisions, that is, by deciding who can access what. In order to make access decisions, a computer system uniquely identifies each user on the system and stores information in named files, each of which belongs to a single user on the system. It would be a potential violation of security if users could access any files at will.

Basic security is supplied through the use of the `login` and `passwd` (password) mechanisms, which identify you to the system and put you in control of your data.

Also included in basic security are access mode bits, which give users some control over which users can access their files.

Enhanced Security

The mechanisms available with the UNIX system enhanced security are designed to protect sensitive information.

Sensitive information must be specially protected according to the rules of a security policy because its unauthorized disclosure, loss, or alteration will cause damage or harm. The enhanced security mechanisms that protect sensitive information are part of the TCB.

Because the information on a computer system can be easily shared and potentially stolen, the TCB has mechanisms that restrict the sharing of information. By controlling access, the TCB protects your files and programs from being seen or accessed by other computer users.

The security policy for the UNIX system requires a relationship between access rules and access attributes:

- access rules allow the TCB to define several distinct levels of authorization
- access attributes provide the mechanism for the TCB to prevent unauthorized access to sensitive information.

NOTE

The levels of authorization defined by the TCB are more complicated and restrictive than the access control provided by the access mode bits.

More specifically, the security policy for a computing system running the UNIX system with enhanced security describes the relationships among five elements.

subject	Subjects cause information to flow among objects or they change the system status. Subjects are represented on the system by processes. Typically, subjects create, read, or write objects.
object	Objects are the parts of a computing system that contain or receive information. Examples of objects are data files, program files, memory, terminals, line printers, disks, tapes, and processes. A process is a subject when it requests an action.
access attribute	The access attributes of a subject or object define its position within the separation scheme that the TCB uses to segregate computer users and information on the computer system.
access rule	The access rules contain the policy that segregates information for the system. The TCB determines whether a subject can access a given object by comparing the access attributes of the subject with the access attributes that are required to access the object. A subject can access an object only when it passes all relevant access checks.
process privileges	Process privileges determine if a subject can perform certain restricted system calls, commands, and functions. Privileges also allow some system calls to override access checks.

Enhanced security is set up hierarchically. A subject can read an object if and only if the subject's level is equal to or higher than the level of the object. A subject can write to an object if and only if the subject's level is equal to the level of the object.

In enforcing the security policy, the TCB assigns access permissions to subjects and objects according to the local security policy instituted by the system administrators, and then uses the access rules to ensure that subjects do not access objects for which the subjects do not have the proper access attributes.

The TCB further restricts the use of certain commands and system calls to subjects (processes) that have the proper privileges.

This limits the ability of users to allow access; the TCB makes access decisions. Security is enhanced because the ability to grant access is enforced by the TCB, not by individual users.

The following sections discuss in more detail the security policy and the access mechanisms that the TCB uses to enforce that policy.

Security Policy

The system's security policy is designed to meet the B2 security criteria established in the "*Trusted Computer System Evaluation Criteria*" (DoD 5200.28.STD, 1985). It consists of the access rules that the system follows, and the exceptions to those rules allowed by privileged processes in the system.

The access rules dictate the limits on non-privileged users of the system; the system's access mechanisms implement the various rules.

The access mechanisms in the UNIX system control the access of subjects to objects, determining whether or not a subject may access a given object, and how. The two access control mechanisms in a UNIX system are

- Discretionary Access Control (DAC), implemented by adding Access Control Lists to the UNIX system file mode mechanism
- Mandatory Access Control (MAC)

When performing access checks, MAC checks are performed first, followed by DAC checks. (Whether or not the checks are satisfied can be modified by the presence of privileges.)

Subjects, Objects, and Access Types

A subject is an active entity that causes information to flow between objects or changes the state of the system or an object. The only subjects in the UNIX system are processes. Since all users are represented in the system as processes, controls on users (subjects) are implemented as controls on processes.

An object is a passive entity that contains or receives information. Objects in a UNIX system are:

- file system entities

  - files
  - directories
  - device special files
  - other special files
- processes
- pipes
- IPC structures
  - messages
  - semaphores
  - shared memory

The access types for a subject to an object fall into three categories:

- read access to an object does not modify the object, but causes information to flow from the object to the subject
- write results in a change to the object, and/or information flow from the subject to the object
- execute/search does not modify the object, but causes information to flow to the subject (as in the results of the search of directory) or causes the creation of a new subject (as in the execution of a file that results in a new process)

Discretionary Access Control (DAC)

Discretionary Access Control is a way of controlling access to objects that is exercised at the discretion of the owner of the object and is enforced by the TCB. For each object owned by a user, the user can designate the access type for other users on the system.

The UNIX system provides the file permission bits (or file mode) mechanism, through which users can grant and deny access to themselves, a group of users to which they belong, and all others on the system. With administrative assistance, users can utilize the group mechanism to effectively grant and deny access to individual users (see *System Administrations'*, "*User Account and Group Management*" chapter, for more details). The UNIX system with the Enhanced Security Utilities installed adds the flexibility of Access Control Lists (ACLs), through which access can be granted and denied to individual users.

ACLs are designed to be compatible with the UNIX file mode scheme. This ACL scheme supports finer control than file permissions by providing the ability for the owner of an object to grant or deny access by other users to the granularity of a single user, while maintaining compatibility with the file permission mechanism. A combination of permission mode bits can be directly translated into an ACL that provides identical protection.

All DAC information may be changed in one atomic operation with the command **`setacl`**, avoiding the possibility of an intermediate insecure state.

ACLs also allow specification of access rights to members of groups as defined to the system in the administrative file **`/etc/group`**.

The number of ACL entries is not limited by the system. The system administrator can set the maximum number of entries per ACL by setting a tunable parameter. (As ACLs get larger, processing gets slower, which induces a practical limit on the number of ACL entries.)

- ACLs are associated with each file system object and IPC object
- ACLs for file system objects are stored in the associated inode
- ACLs for IPC objects are stored in an internal structure associated with the installation of the IPC object.

The **`getacl`** command reports that each ACL has entries corresponding to the file mode permissions for owner, group, and other. ACL entries are described in greater detail in the next section.

A summary of the DAC access check algorithm plus some background ACL information and definitions required to understand the algorithm is presented here.

Basic ACL Entries

An ACL contains all the DAC access information for its associated object. File permission bits are translated into and stored as ACL entries. When a file is created, the permission mode bits and the basic ACL are generated. The basic ACL generated at time of file creation has four entries:

- `user`
- `group`
- `class`
- `other.`

The `owner` mode bits are always equal to a `user` ACL entry for the object's owner.

The `other` mode bits are always equal to the `other` ACL entry, of which there is only one in any ACL.

The `group` mode bits are initially equal to the `class` ACL entry, of which there is only one in any ACL and to the `group` ACL entry for the owning group.

The `class` entry is discussed later in this section.

The basic or initial ACL can be extended by specifying additional `group` and `user` entries. Permissions for multiple groups can be specified in `group` entries, while additional `user` entries can be used to grant or deny access for specific logins.

The ACL can have additional entries based on the `default` ACL entries associated with the directory in which a file is created. These `default` directory ACL entries indicate the ACL entries that are added to any file created in that directory.

ACL Generation for Files

The following describes how an ACL for a file is generated. Whenever a file is created, the system initializes an ACL for the file that contains

- a `user` entry for the owner permissions
- a `group` entry for the owning group permissions
- a `class` entry for the owning group permissions
- an `other` entry for the other group permissions.

Additional entries may be added by the user, or as a result of default entries specified on the parent directory.

The **`getacl`** command reports the entries in the ACL. Each ACL has at least four entries, one each corresponding to the file mode permissions for owner, group, class, and other.

File permission bits for user and group are translated into special cases of these entries:

- the bits representing owner permissions are represented by a `user` entry without a specified user ID
- the bits representing group permissions are represented by a `group` entry without a specified group ID.

In an ACL, there must be one each of these special user and group entries. There may be any number of additional `user` entries and `group` entries, but these must all contain a user ID or group ID, respectively.

There is only one `other` entry in an ACL, representing the permission bits for permissions to be granted to other users. The following is an example of the output of the **`getacl`** command for a file named **`junk`** owned by `user_1` in `group_1` whose permission mode bits are `-rw-r--r--`:

```
# file: junk
# owner: user_1
# group: group_1
user::rw-
group::r--
class:r--
other:r--
```

If `user_2` and `user_3` and `group_2` are added to the ACL by using the **`setacl`** command, **`getacl`** would produce the following output:

```
# file: junk
# owner: user_1
# group: group_1
user::rw-
user:user_2:r--
user:user_3:r--
group::r--
group:group_2:r--
class:r--
other:r--
```

The mode bits on the ACL `class` entry are significant. The `class` entry mode bits are determined by the `group` mode bits for the file. The `group` entry for the owning group and the `class` entry in the basic ACL are identical.

When only a basic ACL exists for the file, the `group` and `class` bits are the same.

When additional users and groups are added to the ACL, the owning `group` bits take on a separate identity from the `class` bits.

If the **`chmod`** command is used to modify DAC permission bits when additional ACL entries exist, it modifies the `owner`, `class`, and `other` mode bits.

In the case of the DAC group permission bits, it is the `class` bits that are modified and not the owning `group` bits in the ACL entry. The only way to change ACL entries (except for the ones representing `owner` and `other`) is by using the **`setacl`** command.

The purpose of the `class` entry bits is to define the maximum permissions available to the users and groups that may be added to the ACL. For example, if the group permission bits for a file are `r--,` the output of the **`getacl`** command would show a `class` entry in the ACL with those same permissions associated with it. All additional users and groups will effectively have no permissions in excess of `r--` regardless of what permissions are indicated in their ACL entry. The specified permissions in the ACL entry for a user or group can only serve to further restrict permissions since the `class` entry derived from the `group` permission bits effectively sets the upper bound for the permissions on additional users and groups.

In this example of an ACL with additional entries, if `user_2` were added to the ACL with `r-x` permission bits, the ACL displayed by the **`getacl`** would look as follows:

```
# file: junk
# owner: user_1
# group: group_1
user::rw-
user:user_2:r-x      #effective:r--
user:user_3:r--
group::r--
group:group_2:r--
class:r--
other:r--
```

However, the effective permissions for `user_2` would be `r--` as determined by the mode bits in the `class` entry.

DAC Access Check Algorithm

The DAC check algorithm is shown in Figure 2-6.

```
if effective uid of process matches owner id on object
    if requested access mode matches bits set in
            the user entry representing the owner
        then the requested access is granted

else if effective uid of process matches the uid in an
                 additional user entry
    if requested access mode matches bits set in that
user entry
                  and matches bits set in the class entry
        then the requested access is granted

else if any group in the group set of process matches
the owner gid or
            the gid of any additional group entry
    if requested access mode matches bits set in any
       combination
                of one or more group entries
                and matches bits set in the class entry
        then the requested access is granted

else if requested access mode matches a bit set in the
other entry
    then the requested access is granted
    else the requested access is denied.
```

Figure 2-6. DAC Access Check Algorithm

In the step that checks for a match against the group, the permissions for all `group` entries are `OR`'d together. For example, if a user is a member of groups A and B, and requests read/write access to a file which allows read access to group A and write access to group B, the requested read/write access will be granted.

Mandatory Access Control (MAC)

Mandatory Access Control is a means of controlling access to objects that is controlled by the administrator of the system and is enforced by the TCB. A non-privileged user cannot affect or bypass this control. It is independent of the object owner's ability to grant discretionary access.

MAC Concepts and Definitions

The UNIX system MAC associates a label with each object and subject in the system. Labels contain a sensitivity level or just level, indicating the sensitivity of the data in the object or the clearance of the associated subject.

An ASCII name, defined by the system administrator, is associated with each classification and each category for easier use and recognition by administrators and general users. Classification names may be something like “top_secret” or “unclassified.” Categories, which are analogous to need-to-know areas, might have names such as “nuclear” or “salaries.” The system supports 256 classifications and 1024 categories.

A unique association of a named classification with zero or more named categories is called a fully qualified level. For easier reference, an alias for the fully qualified level can be defined. To be recognized and used on the system, the set of all fully qualified levels must be defined to the system by associating a unique numerical identifier with each fully qualified level to be recognized by the system. This is known as a level identifier, or LID.

The mappings of fully qualified levels to LIDs to aliases are kept in an administrative database file. As delivered, the UNIX system with the Enhanced Security Utilities installed has several reserved alias names, some of which have pre-defined fully qualified levels associated with them. The reserved alias names with which there are associated pre-defined fully qualified levels are shown below:

- `USER_PUBLIC`
- `USER_LOGIN`
- `SYS_PRIVATE`
- `SYS_PUBLIC`
- `SYS_OPERATOR`
- `SYS_AUDIT`
- `SYS_RANGE_MIN`
- `SYS_RANGE_MAX`

`USER_LOGIN` defines a default login level for users of the system; this may be changed as appropriate for a given installation of the system.

`USER_PUBLIC` provides a level for public user commands and files.

`SYS_PRIVATE`, `SYS_PUBLIC`, `SYS_OPERATOR`, and `SYS_AUDIT` are reserved for administrative users and programs and help protect the UNIX system TCB.

`SYS_RANGE_MIN` and `SYS_RANGE_MAX` are the low and high level bounds for the system. No users log in at these levels; they are only for system use.

Reserved alias names without associated pre-defined fully qualified levels are `SYS_LOGIN_HIGH` and `SYS_LOGIN_LOW`. These are to be set by the system administrator and establish the high and low bounds for the system's login level range.

MAC Access Rules

The objects under MAC control are:

- File system objects
  - regular files
  - device special files
  - links
  - FIFOs (named pipes)
  - regular and multilevel directories
- unnamed pipes
- IPC objects
  - shared memory
  - semaphores
  - message queues
- processes.

Multilevel directories and device special files require special handling for MAC. They are discussed separately in subsequent subsections of this document.

Rules for Objects

The following general rules apply for all other objects.

- To read an object, the subject's clearance must dominate the level of the object.
- For one level to dominate another, two conditions must be met:
  - the classification of the first must be at the same or a higher hierarchical level; and
  - the set of categories of the first must be a superset of the set of categories of the other.
- IPC objects and pipes obey a more restrictive dominance rule; for IPC objects and pipes, the level of the subject must equal that of the object before read is allowed.
- To write an object, the subject's level must be equal to that of the object.

- To search/execute an object, the subject's clearance need only dominate that of the object. This is implemented following the same rules as read. Search/execute access does not apply to pipes or IPC objects.
- To access objects contained in multilevel directories, the subject's clearance must equal that of the object for all forms of access.

Establishing the Levels of Objects and Subjects

There are two situations in which the level of an object may be set:

- creating an object
- administrative change.

Creating and object is a special case of write. The level of a newly-created object is set equal to that of the process that creates it.

An administrator with appropriate privileges can change the level of an existing object. The administrator must take care that, when changing the level of an object, information is not unwittingly being downgraded.

The level of a subject (that is, process) is generally set by the parent process. The two general cases are:

- At login, the user's process is created on behalf of the user by the login system process. The user's level is determined by the login process: it is either explicitly specified to `login` by the user as part of the **`login`** command, or, if the user chooses not to specify a level, **`login`** takes it from a system database containing default levels for all users of the system.

  After this and all other user parameters are established, **`login`** provides a process for the user with the correct level. If either the specified or default level is outside of the system's current login level range, the login fails. (Checks against login level range are only made at login time.) If the specified or default level is not a level that is authorized for that user, the login fails.
- A child process (created by `fork`/`exec` sequence) or a new process image (created by `exec`) of an existing process takes the level of the parent or invoking process. The only exception is in the case of an appropriately privileged process, which may create a new process at a different level than itself (as in `login`).

An appropriately privileged process may change its own level. Only processes that are part of the TCB are trusted do this.

The Kernel, Access Control, and Security

The following sections discuss the architecture of the UNIX system kernel and how the Enhanced Security Utilities fit into the architecture of the kernel.

The UNIX system kernel defines an interface between user applications and the system hardware including functions that supply system services to users and functions for internal housekeeping, hardware management, and all the activities shown in Figure 2-7.

The entire kernel is included in the TCB. The kernel mediates all security-relevant decisions on the system in accordance with the security policy. The kernel implements the reference monitor concept defined in the *Trusted Computer System Evaluation Criteria*.

The kernel is organized into distinct modular service areas that enhance the security of the kernel code and facilitate testing of the kernel.

Kernel code is compiled into a single executable file, named **`/stand/unix`** by default, that is loaded and run at boot time. The entire kernel is always resident in primary memory, runs in its own address space, and is protected from all access by user processes. Kernel code runs in a mode that allows it to execute privileged hardware instructions.

The following figure shows the major subsystems of the kernel. Note some functionality provided by each subsystem may actually be executed in another subsystem.

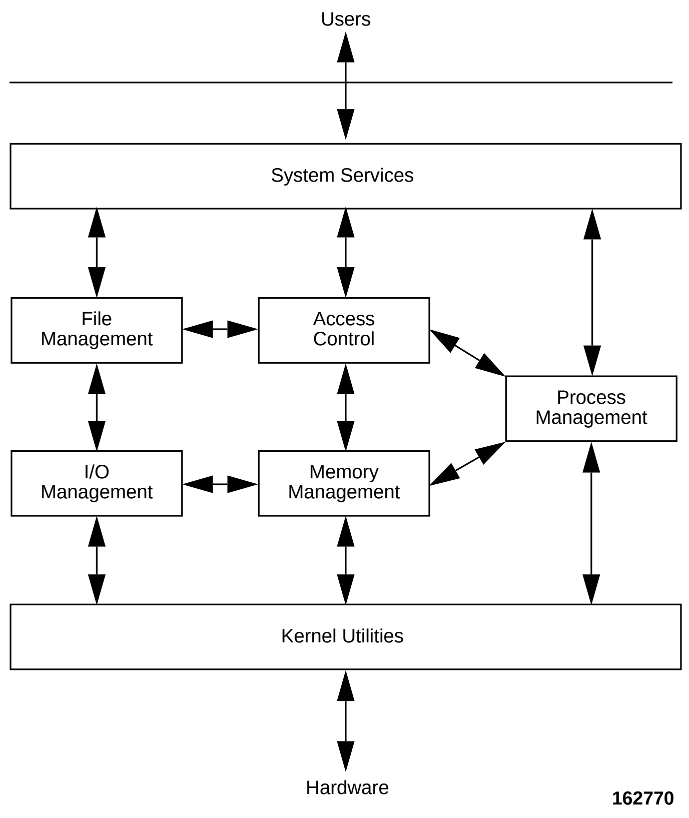


Figure 2-7. Kernel Architecture

- The process management subsystem manages programs in execution (processes). The process is the only subject in the UNIX system.
- Protection of the TCB software is through the appropriate use of the access controls and privilege mechanisms.

- The access control subsystem provides control and monitoring of access to files, processes, and privileged operations. It provides the discretionary and mandatory access control, privilege control, and auditing functionality.
- The I/O management subsystem manages input and output. It includes the STREAMS mechanism (see *Device Driver Programming: STREAMS I/O Modules and Drivers*) and device drivers.

  Device drivers may be classified into two basic types:

  - Character drivers read/write a character at a time (for example, terminal drivers). Such drivers typically use the STREAMS mechanism.
  - Block drivers read and write in chunks (for example, disk drivers). Disk drivers might read and write in chunks of 512 bytes, for example

  In the UNIX system, I/O devices are file system objects; from user level, devices are treated like files for basic I/O functions such as open, close, read, and write.
- The memory management subsystem manages virtual and physical memory. It presents a virtual memory (VM) interface to programmers. It protects users from each other and protects kernel address space from all access by user processes, thus making the kernel tamper resistant.
- The VM system uses demand paging to make efficient use of physical memory. It also unifies common memory and I/O operations by allowing memory-mapped files: a mapping can be established between a file and an area of primary memory; after the mapping is set up, standard memory operations such as assignment and bit sets and clears can be used.

  Memory mapping allows both a conceptual simplification of some programs as well as a large performance improvement for some devices (for example, bit-map terminals). The VM system ensures that areas of memory are cleared upon allocation for use by processes.
- This subsystem provides functionality needed by user programs but not necessarily used directly by the kernel. The system services subsystem manages system initialization and termination, clock and timer services, trap handling, and customized local system call dispatching, support for system dumps, and declarations supporting system calls, stubs, kernel debugging, and system-specific character strings.
- The kernel utilities subsystem controls the interaction of the hardware and the kernel and provides other miscellaneous utilities used by other subsystems, such as: double precision arithmetic operations used by the high-resolution timers in the System Services Subsystem; math accelerator interface routines; recording of significant system events; bitmap operations; system error message support; kernel performance statistics; machine dependent routines (for example, DMA access, memory allocation); and, other miscellaneous subroutines.

User Level

All user code and UNIX System code that is not part of the kernel runs at user level. This includes all commands, including administrative commands and command interpreters ("shells"). User-level code also includes all library functions, which are described in Section 3 of the online *Operating System API Reference.*

Some of this code is considered part of the TCB (though not part of the kernel). These user-level program and data files on the system that are part of the TCB have associated properties that allow the kernel to control both their run-time behavior and the ability of users to access them.

Every command and data file included in the TCB has appropriate discretionary and mandatory access controls, and process privilege requirements associated with it that prevent unauthorized use or modification.

The following attributes are given to user-level TCB files:

- appropriate privileges required for executing the file (commands and shell scripts only); programs that execute sensitive system calls have associated privileges that can be inherited by a privileged process
- MAC label for all commands, data files, and the directories that contain them; programs, data files, and directories that are strictly for administrative use are given a label of `SYS_PRIVATE;` programs, data files and directories for use by both administrators and users are given a label of `SYS_PUBLIC;` all mount points are given the label `SYS_PUBLIC.`
- DAC controls; all binary programs with a label of `SYS_PRIVATE` are not given write permissions; all files/directories with `SYS_PRIVATE` or `SYS_PUBLIC` labels are not given write permission for other.

All users, including administrators, log in as unprivileged users; the initial user process has no privilege associated with it. Some users designated as administrators or operators must execute commands that require one or more process privileges. User (unprivileged) processes gain appropriate privilege through the **`tfadmin`** command and the Trusted Facility Management database.

Several distinct levels of authorization are created through the proper assignment of process privileges according to the least privilege rule and the separation of duties that is accomplished through the TFM database. This ensures both that privileged processes run only with the privilege(s) required for the actions they are authorized to perform, and that unprivileged processes cannot perform privileged actions. Users acquire privilege in one of three ways:

- logging in to the system using the privileged user ID, usually `root`, but configurable by the administrator
- by spawning a process with an effective user ID equal to the privileged user ID; this is usually done by executing a file owned by the privileged user ID that has the setuid-on-execution bit of the file mode set
- by executing a file with privileges set on it.

System Calls

System calls form the interface between user level and the kernel. The system calls are the only way of requesting a service from the kernel. All user and administrative programs must eventually make system calls to perform security-relevant functions such as a file operation or I/O request. Many system calls can be made indirectly through the libraries.

In the C language, there is no syntactic way to distinguish a system call from any other function call. Neither system calls or library routines are known to the C compiler.

What distinguishes the system calls is that the code that implements them is part of the kernel and runs in kernel mode; a process must undergo a fundamental change of environments (also called a context switch) when it begins a system call. The compilation system knows how to initiate the change of environments. It arranges the arguments to be passed into the kernel, then executes the instruction that traps into the kernel. Note, however, that compilers are not part of the TCB or kernel; they only know how to set up for a context switch and execute the instruction that traps into the kernel, which then performs the environment change.

System call arguments are put on a stack in user address space; these might be data or pointers to data. When control is passed to the kernel, it knows how many arguments to take off the stack and copies them to kernel address space. The kernel then performs the operation(s) indicated by the system call on behalf of the user.

Kernel entry points are defined in the software in the kernel structure `sysent` and described in Section 2 (System Calls) of the online *Operating System API Reference.*

Processes

A process is a program in execution. A compiled program typically has a `text` section to hold its executable code, a `data` section to hold its initialized data, and a `bss` section to hold its uninitialized data. The program begins execution in user space. When the program is executed, the kernel loads the program's sections as it needs them and creates a stack in user space to manage the process' subroutine calls. (The kernel keeps its own stack to manage its function calls.)

A process has a unique ID number (its process ID or PID), a sensitivity level, and other attributes (such as the user ID and group ID of the process) that the kernel uses to make access control decisions. The kernel compares these attributes to an object's access control attributes to determine access.

If a program calls a system library function, execution branches to the text segment of that function. All operations take place in user address space; that is, no objects (such as files, and so on) are maintained in the libraries.

Library functions are typically held in a shared library, which is loaded once and can then be used by any program that calls its functions. Shared libraries save space in primary memory because a function is loaded once in the entire system instead of once for each program that calls it. Most, but not all, library functions perform system calls.

The library object file(s) are protected with appropriate discretionary and mandatory access controls to prevent tampering by unauthorized users. No privileges are set on the library files; if a piece of library code makes a system call that requires privilege, the user process running the library code must have appropriate privilege to perform the requested operation, or the call will fail.

If a program (or library function) makes a system call, execution branches to a small piece of library code that knows how to set up for an execution switch and trap into the kernel. The kernel then performs a context switch on the process, after which the process is executing in kernel mode. In kernel mode, the process is executing kernel code on behalf of the user that made the system call.

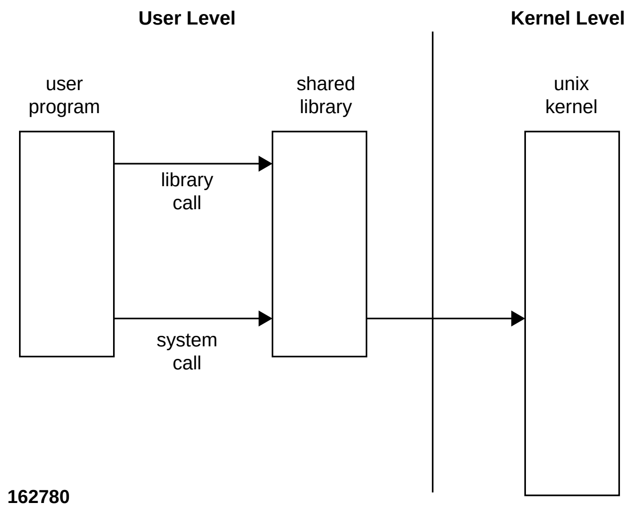


Figure 2-8. User and Kernel Level

The **`fork(2)`** and **`exec(2)`** system calls start new processes. The `fork` system call creates a copy of the calling process and puts it into execution. The calling process is known as the parent process; the new process is known as the child.

The `exec` system call overlays the calling process and replaces it with a new process. The following is the algorithm for process creation:

```
. . .
if (fork ()) {
        /* child returns here */
        exec (new_program_name  . . .) /* start new program */
        /* this location is never reached */
}
/* parent continues here */
. . .
```

Before the `fork`, this code executes in a single process. If the `fork` succeeds, a new process is created and begins executing the code following the `if` statement; at this point the code is being executed in two processes. The `exec` transforms the child process into a new one; at this point the parent continues executing the code at the point indicated and the child is executing the new program.

The child process inherits the parent process' security level and the privilege sets associated with the parent.

Every process has both a maximum set and working set of privileges associated with it. Similarly, every executable file has both a fixed set and inheritable set of privileges associated with it.

The working and maximum privilege sets of the parent are replicated in the child. However, once the new process begins execution (that is, the `fork` succeeds), changes to the privilege sets of the parent do not affect the child.

Upon executing the `exec` system call, the working set of the new child is initialized to the maximum set. The new process's maximum and working sets are determined by taking the logical intersection of the maximum set of the old process and the inheritable set of the executable file; the logical union of this resulting set and the fixed set of the executable file form the new maximum set of the new process.

Privileges will be added to and removed from the new process's working set as the process executes and performs operations requiring one or more of the privileges in the maximum set. The new process's maximum and working sets are determined by taking the logical union of the maximum set of the old process and the fixed set on the executable file.

Privileges will be added to and removed from the new process's working set as the process executes and performs operations requiring one or more of the privileges in the maximum set.

The label of the process remains the same across the `exec` system call, that is, the new process has the same label as the old process.

Interprocess Communication (IPC)

Interprocess communication (IPC) allows a process to set up a memory region that can be used to communicate with other cooperating processes. These memory regions have user IDs, group IDs, permissions, ACLs, and levels just like file system objects.

There are three types of IPC objects:

- shared memory
- messages
- semaphores.

The interface to these objects allows the user to treat them like file system objects, but they are not part of any file system. That is, IPC objects are accessed through identifiers (as in a flat file system), but do not have inodes like files. IPC objects are maintained entirely in the kernel.

Most IPC access requires both read and write. For example, when a message is read, it is destroyed, access checks are made appropriately, and the mandatory access policy for writing is enforced for the operation.

While setting up and accessing IPC objects may require privilege, there are no privileges associated directly with IPC objects. Privileges are handled at the command and system call level. This is analogous to the way privileges are handled by commands and system calls for creating, modifying, and deleting file system objects; similar access restrictions apply to IPC objects.

Note that although pipes provide a kind of inter-process communication facility, they are not considered IPC objects in the same sense as the above-described mechanisms, since the functionality they provide is more limited.

Signals

Signals provide a means of communication between a sending process acting as a subject and a receiving process, the object. Sending a signal is regarded as a write operation: for unprivileged processes, the level of the subject process must equal the level of the object process. An appropriately privileged process may bypass the MAC restriction.

This chapter has described some basic principles of the UNIX operating system. The following chapters in this manual will help you apply these principles.

3
Basics for UNIX System Users

3
Basics for UNIX System Users

Introduction

This chapter describes how to use the UNIX system. Specifically, it lists the required terminal settings, and explains how to use the keyboard, obtain a login, log on and off the system, and enter simple commands.

To establish contact with the UNIX system, you need:

- a terminal
- a login name that identifies you as an authorized user
- a password that verifies your identity
- instructions for dialing in and accessing the UNIX system if your terminal is not directly connected to the computer
- the Secure Attention Key (SAK) from the system administrator.

This chapter follows the notation conventions used throughout this guide. For a description of them, see Chapter 1, "*Introduction.*"

The Terminal

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

- video display terminals
- printing terminals.

The video display terminal shows input and output on a display screen; the printing terminal, on continuously fed paper. In most respects, this difference has no effect on the user's actions or the system's responses. Instructions throughout this book that refer to the terminal screen apply in the same way to the paper in a printing terminal, unless noted otherwise.

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 terminal 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 on any terminal before trying to log on the UNIX system.

1. Turn on the power.
2. Set the terminal to on-line or remote operation to ensure the terminal is under the direct control of the computer.
3. Set the terminal to full duplex mode. Full duplex is a communication protocol in which both sides send and receive simultaneously. The UNIX system operates in full duplex. A full duplex connection lets you send information to the UNIX system even while it is sending data to you.
4. If your terminal is not directly connected to the computer, make sure the modem you are using is set to the full duplex mode.
5. Set character generation to lowercase.
6. Set the terminal to no parity. Parity is used by some systems to do error checking. The UNIX system does not use 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 4800 sends and receives approximately 480 characters per second (cps).

   Depending on the computer and the terminal, baud rates between 300 and 19200 are available. (Some computers are capable of processing characters at higher speeds.)

Keyboard Characteristics

Even though there is no standard layout for terminal keyboards, 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 of a typical ASCII terminal is similar to that of a typewriter, but contains a few additional keys for functions such as interrupting programs. The keys may be divided into the following groups:

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

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

Typing Conventions

To interact with the UNIX system, you must 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 or two keys. Table 3-1 lists these conventions.

Table 3-1. UNIX System Typing Conventions

Key	Function
<$>	system's command prompt (your cue to issue a command)[1]
<BACKSPACE>	erase a character
<CTRL>	used with other characters to perform controlling actions on lines of typing
<CTRL><h>	erase a character
<@>	erase an entire line
<BREAK>	stop execution of a program or command
<DEL>	delete the current command line
<ESC>	when used with another character, performs a specific function (escape sequence)
	when used with the **vi** editor, ends text input mode and returns to the command mode
<RETURN>	send the current line to the system for execution
<CTRL><d>	stop input to the system or log off[2]
<CTRL><h>	backspace for terminals without a <BACKSPACE> key

Table 3-1. UNIX System Typing Conventions (Cont.)

Key	Function
<CTRL><i>	horizontal tab for terminals without a <DEL> key
<CTRL><s>	temporarily stop output from printing on the screen
<CTRL><q>	resume printing on the screen after it has been stopped by the <CTRL><s>key

1. The prompt character may differ on your system.

2. Characters shown as <CTRL>-*char*, where *char* is a letter of the alphabet, are called control characters (pronounced "control-*char*"). To enter a control character, hold down the <CTRL> key and press the specified character key.

NOTE

The key(s) associated with each function are default values; in most cases different keys could be chosen to perform the same function. Detailed explanations of several of the keys are provided on the next few pages.

The Command Prompt

A command prompt is a character that appears on your screen when the system is waiting for instructions from you. When a prompt appears, type a command after it and press the <RETURN> key.

The default command prompt for the UNIX system is the dollar sign ($), but if you prefer to use another character (or string of characters), you can change your prompt. (Your system administrator may have already changed this default.)

Correcting Typing Errors

There are several methods of deleting text for the purpose of correcting typing errors:

- the @ (at) sign erases the current line
- the <BACKSPACE> key and <CTRL><h> both erase the last character typed.

All these signs and keys are defaults; the functions they provide may be reassigned to other keys. (For instructions, see "*Reassigning the Delete Functions*" later in this chapter.

Deleting the Current Line: The @ Sign

The @ sign deletes 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; you must retype the correct command on the next line.

The @ sign works only on the current line; be sure to press it before you press the <RETURN> key if you want to delete a line. In the following example, a misspelled command is typed on a command line; the command is canceled with the @ sign, and the correct command is retyped on the next line:

```
whooo@
who <RETURN>
```

Deleting Last Characters Typed: <CTRL><h> & <BACKSPACE>

The <BACKSPACE> key and <CTRL><h> both delete the character(s) last typed on the current line. When you type either, the cursor backs up over the last character and lets you retype it. This is an easy way to correct a typing error.

You can delete as many characters as you like by typing a corresponding number of <BACKSPACE> keys or <CTRL><h>characters. For example, in the following command line, two characters are deleted by typing two <BACKSPACE> keys.

```
dattw <BACKSPACE> <BACKSPACE> e <RETURN>
```

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

Reassigning the Delete Functions

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 as soon as you log off.

If you want to use other keys regularly, you must specify the reassignment in a file called **`.profile.`**

Remember:

- 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 <BACKSPACE> key to the <#> sign, you will no longer be able to use the <BACKSPACE> key to erase characters. And, you will not have two keys that perform the same function.
- These 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 the “Line Editor (ed) Tutorial” later in this book) allows you to delete text with the same key you use to correct errors on a shell command line (as described in this chapter). If you reassign the erase function to the # sign, you will have to use the # sign to erase characters while working in the **`ed`** editor, as well. The <BACKSPACE> key will no longer work.
- Any reassignments you have specified in your **`.profile`** do not become effective until after you log in. If you make an error while typing your login name or password, you must use the <BACKSPACE> key to correct it.

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

Using Special Characters as Literal Characters

What happens if you want to use the literal meaning of a special character? Because the UNIX system's default behavior interprets 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.

You can do this with the backslash (\) character. When you type a \ before any special character, you 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:

He bought three pounds @ $.05 cents each.

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 all the words before the @ sign and print your sentence as follows:

```
$.05 cents each.
```

To avoid this, type your sentence as follows:

```
He bought three pounds \@ $ .05 cents each.
```

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. Because 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. (It communicates in full duplex mode.)

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 Commands

To stop the execution of most commands, simply press the <BREAK> or <DELETE> key.

The UNIX system will stop the program and print a prompt on the screen. This is its signal that it has stopped the last command from running and is ready for your next command.

Using Control Characters

Locate the control key on your terminal keyboard. (It may be labeled <CTRL> or <CONTROL> 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 typing. Commands entered in this way are called control characters.

Some control characters perform mundane tasks such as backspacing and tabbing. Others define commands that are specific to the UNIX system. For example, one control character (<CTRL><s> by default) temporarily halts output that is being printed on a terminal screen.

To type a control character, hold down the <CTRL> key while pressing the appropriate alphabetic key. Because control characters are entered by pressing both keys simultaneously, they're represented, in this book, by the <CTRL>-*char* icon, where *char* is the appropriate letter of the alphabet. For example, <CTRL><s>is an instruction to press the <CTRL> and <s> keys simultaneously.

The two functions for which control characters are most often used are to control the scrolling of output on the screen and to log off the system. To prevent information from scrolling off the top of the screen on a video display terminal, type <CTRL><s>; the scrolling will stop.

When you are ready to read more output, type <CTRL><q>and the scrolling will resume.

To log off the UNIX system, type <CTRL><d>.

It is important that you log off whenever your terminal is going to be unattended. If you fail to do this, anyone with access to your terminal can use your login session to read or write any files that you can read or write. (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. For example, if your keyboard does not have a <BACKSPACE> key, you can enter <CTRL><h> instead. Or, if you don't have a <TAB> key, you can insert a tab by typing <CTRL><i>. (See *"Problems When Logging In"* for information on how to set the <TAB> key.)

Obtaining a Login Name

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

A login name is the name by which the UNIX system verifies that you are an authorized user of the system. You must enter it every time you want to log in. (The expression "logging in" is used because the system keeps a log of dates and times when users request access to the system.)

To get a login name, set up a UNIX system account through your local system administrator. There are few rules governing your choice of a login name:

- It should be three to eight characters long.
- It can consist of uppercase and lowercase letters, numbers, and the underscore character (_), but it cannot start with a number.

When selecting your login name, keep in mind that UNIX is case-sensitive and, therefore, distinguishes between uppercase and lowercase letters. Popular choices for login names include initials, last names, and nicknames.

Your login name will have a default MAC level assigned to it. Some users will have additional security levels that they can access. If you have access to security levels other than your default level, your system administrator will tell you what they are. Complete information on the implications of using more than one security level can be found in Chapter 14, "*Managing Files Securely*".

Communicating with the UNIX System

NOTE

This section assumes you will be using a terminal that is wired directly to a computer or one that communicates with a computer over a telephone line. Although it describes a typical procedure for logging in, the instructions it gives may not apply to your system since there are many ways to log in to a UNIX system over a telephone line. For example, security precautions on your system may require that you use a special telephone number or other security code. For instructions on logging in to your UNIX system from outside your computer installation site, consult your system administrator.

To log in, you must:

1. Turn on your terminal. If you are directly connected, skip the next step.
2. If you are going to communicate with the computer over a telephone line, you must establish a connection by entering the telephone number that connects you to the UNIX system.

   You'll see one of three messages on your screen:

`BUSY`	The circuits are busy; try dialing again.
`NO ANSWER`	This usually means the system is inoperable because of mechanical failure or electronic problems. Check the connections between your terminal, modem, and phone line and try dialing again.
`ONLINE`	The system is accessible.

If you are not accessing the computer via telephone line, ask your system administrator how to gain access.

3. If accessing your computer requires that you enter a Secure Attention Key (SAK), see “*Getting a Login Prompt with the Secure Attention Key*” later in this chapter for more information. Otherwise, go to the next step.

4. The `login:` should appear.

   If the `login:` prompt is not on your screen, press the <RETURN> key and try again.

   A series of meaningless characters may appear on your screen. This means 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 the <BREAK> or <RETURN> key; this signals the system to try another speed. If the UNIX system does not display the `login:` prompt within a few seconds, press the <BREAK> or <RETURN> key again.

NOTE

Don't press the <BREAK> key, if this is your SAK. Instead, just press the <RETURN> key.

Getting a Login Prompt with the Secure Attention Key

On many computer systems, a user's password is vulnerable whenever it is typed. Because passwords are typed over ordinary, insecure data channels, malicious users can steal passwords by using a spoofing program. A spoofing program is a program created by a malicious user of the system to trick other users into believing that it is the system's login program. When passwords are entered by unsuspecting users, the spoofing program records them and passes them on to the user who wrote the program. With these stolen passwords, this user can gain access to other users' accounts and files.

To protect passwords, this version of the UNIX operating system uses a secure communications channel, called a trusted path, whenever users enter their passwords. The trusted path prevents malicious users from employing spoofing programs or other devices to gain users' passwords.

You can get a login prompt only after entering the Secure Attention Key (SAK) to establish a trusted path between your terminal and the main computer. The trusted path is in place only during login processing. Whenever you want to log on, you must enter the SAK and obtain a new trusted path. (Remember that you can change your password only during login. This ensures that your password is protected when you change it.)

NOTE

If you ever get a login or password prompt without entering the SAK or if you notice any other unusual behavior, report the incident to your system administrator. Your system may not be configured correctly, or there may be a spoofing program on the system. Do not log in until you have talked with your administrator.

To get a `login:` prompt by using the line drop signal as a SAK, either turn the terminal off and then back on, or press a combination of keys. On AT&T 630 terminals, for example, pressing <CTRL><SHIFT><BREAK> sends a line drop signal from your terminal to the host computer.

Once the login prompt appears, you have a limited amount of time in which to log in. There is usually enough time for a second login attempt if you make a mistake the first time. If you mistype your login name or password, you should be able to get a new login prompt by typing <RETURN>, because the trusted path is still in effect for the terminal.

If you do not complete the login process while the trusted path is in effect, you will not get a response from the computer when you type <RETURN>. You must then enter the SAK again to get another login prompt.

If you enter the SAK while you are logged on to the system, your current login session is ended. All currently open connections to the terminal can no longer be used for I/O or control options to the terminal. You can use the terminal only by starting a new login session. Be careful not to enter the SAK by mistake.

The SAK may also be the break signal or a control character; your system administrator can tell you what the SAK is for your terminal. (Different terminals may have different SAKs and the administrator can redefine the SAK for a terminal; if you don't get a login prompt when you enter a SAK, see your administrator.)

CAUTION

Do not attempt to guess the SAK; if you do, you'll be vulnerable to spoofing programs and other attacks on your password. If you forget the SAK or have not been told what it is, see your system administrator.

Be sure you get any information about the SAK from a reliable source. If a malicious user gives you incorrect information about the SAK, your password can be stolen.

Login Procedure

When the login: prompt appears, always enter the SAK before logging in, even if the `login:` prompt is already displayed on the terminal. Entering the SAK ensures that you are not subject to a spoofing program. When the `login:` prompt appears after you enter the SAK,

1. Type your login name and press the `<RETURN>` key. For example, if your login name is `starship,` your login line will look like this:

   ```
   login: starship <RETURN>
   ```

2. If you make a mistake while typing your login name, you may correct it with the `<BACKSPACE>` key or the `@` character, as discussed earlier.

NOTE

> Remember to type in lowercase letters. If you use uppercase when you log in, the UNIX system will expect you to continue using uppercase characters exclusively for the rest of your login session and it will produce output only in uppercase.

3. This will begin the process to log you in at your default security level. You may be authorized to log in at security levels other than your default. There are many implications to using more than a single security level; complete information can be found in Chapter 14, "*Managing Files Securely*". If you are authorized to log in at several security levels, you may specify, with the **`-h`** option, the level to be assigned to your login process. For example, if `blue` is one of the levels to which you are allowed access, you can log in at that level by typing:

   ```
   login: -h blue starship <RETURN>
   ```

4. You can also change your default level to any level for which you are authorized. You do this by specifying the **`-v`** option. For example, if `blue` is one of the levels to which you are allowed access, you can change your default level (so that all subsequent logins will default to that security level) by typing:

   ```
   login: -v blue starship <RETURN>
   ```

5. Next, the system prompts you for your password. Type your password and press the `<RETURN>` key.

   For security reasons, the UNIX system does not echo (that is, print) your password on the screen.

6. If you make a mistake while typing your password, you may correct it with the `<BACKSPACE>` key or the `@` character, as discussed earlier.

7. 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; news items might include announcements of new system tools.)

8. The system will print the date and time of your last login, together with the level at which you logged in and an indication of what terminal you used. You should check this information. If it does not agree with your

recollection, it may be a sign that someone else has used your login. In that case, you should immediately change your password (as described below) and inform the login administrator.

When you have logged in, your screen will look something like this:

```
login: starship <RETURN>
password:
Last login: Wed Dec 4 11:14:22 on term/21 at level blue
UNIX system news
$
```

9. 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: starship <RETURN>
   password:
   login incorrect
   login:
   ```

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 depends on your keeping your password private.

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

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

   Passwords should be constructed to meet the following requirements:

- Each password must have from three to eight characters; the length of your password will be defined by your system administrator. If no specific length is assigned, the length defaults to six characters. The system ignores any characters that exceed the eight-character limit.
- 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. To verify that your password has been entered correctly, 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>
$
```

In addition to having temporary passwords that expire immediately, your system administrator can arrange for all passwords to "age" and eventually expire. If that happens, you must select a new password as in the example above.

Even if password aging is not used on your system, it is a good security practice to change your password whenever you think someone might have learned what it is. To change your password:

1. Type **`-p`** at the `login:` prompt, before your login name. Your login line will look like this:

   `login:` **`-p`** `starship <RETURN>`

2. Once again, the system will ask for your old password to verify who you are and then ask you to type your new password twice.

Problems When Logging In

A terminal may act peculiar when it is not configured properly. 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, check the following and then try logging in again:

keyboard	keys labeled `<CAPS>`, `<LOCAL>`, `<BLOCK>`, and so on should not be enabled (put into the locked position); disable these keys by pressing them.
modem	if your terminal is connected to the computer via telephone lines, verify that the baud rate and duplex settings are correctly set.
switches	some terminals have several switches that must be set to be compatible with the UNIX system; be sure these switches are set properly.

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

Table 3-2 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.

Table 3-2. Troubleshooting Problems When Logging In

Problem[1]	Possible Cause	Action/Remedy
Meaningless characters	UNIX system at wrong speed	Press `<RETURN>` or `<BREAK>`
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 `<CAPSLOCK>`) is enabled	Press `<CAPSLOCK>` (or `<LOCK>`) key 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`**[2]
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 and try logging in again
Communication link (connection to UNIX system) is repeatedly dropped	Bad telephone line or bad communications port	Call system administrator
Cannot get login prompt after making connection and pressing the `<RETURN>` key	Did not enter SAK	Enter SAK; if SAK is unknown, call system administrator

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

2. Typing **`stty -tabs`** corrects the tab setting only for your current computing session. To ensure a correct tab setting for all sessions, add the line `stty -tabs` to your **`.profile`**.

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 the `<RETURN>` key. The UNIX system accesses a program called **`date,`** executes it, and prints its results on the screen, as shown below.

```
$ date <RETURN>
Thu Jul 18 14:49:07 EDT 1991
$
```

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

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

```
$ who <RETURN>
starshipterm/00Jul 188:53
mlf  term/02Jul 188:56
jro  term/05Jul 188:54
ral  term/06Jul 188:56
$
```

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 may 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 really be logged off the system.

If you enter the SAK after you log off, the UNIX system will display the `login:` prompt again. The following example assumes you have a direct connection to your terminal and that `<CTRL><SHIFT><BREAK>` is the SAK.

```
$<CTRL><d>
$<CTRL><SHIFT><BREAK>
login:
```

This shows that you have logged off successfully and started a new login session.

If you do not have a direct connection to your terminal (you may communicate with your terminal via modem), pressing <CTRL><d> or <CTRL><SHIFT><BREAK> will terminate your login session. To log in again, see the section, "*Login Procedure*" in this chapter.

NOTE

Always log off the UNIX system by typing <CTRL><d> followed by the SAK before you turn off the terminal or hang up the telephone. If you do not, you may not really be logged off the system. Because anyone who gains access to your login session can use the system as if they were you, it is extremely important that you log off whenever you are going to leave your terminal unattended. To ensure that you're logged off, it's a good practice to enter the SAK.

4
Using the File System

4
Using the File System

Introduction

To use the UNIX system effectively you must be familiar with file system 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 file systems.

The first two sections (*"File System Structure"* 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.

File System Structure

The file system is made up of a set of ordinary files, special files, symbolic links, and directories. These components provide a way to organize, retrieve, and manage information electronically. Chapter 2 "*What Is the UNIX System?*" chapter 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 symbolic link is a file that points to another file.
- 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 hierarchical tree structure. Figure 4-1 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. There is no limit to the number of files and directories you may create in any directory that you own. Nor is there a limit to the number of layers of directories that you may create. You can organize your files in a variety of ways, as shown in Figure 4-1.

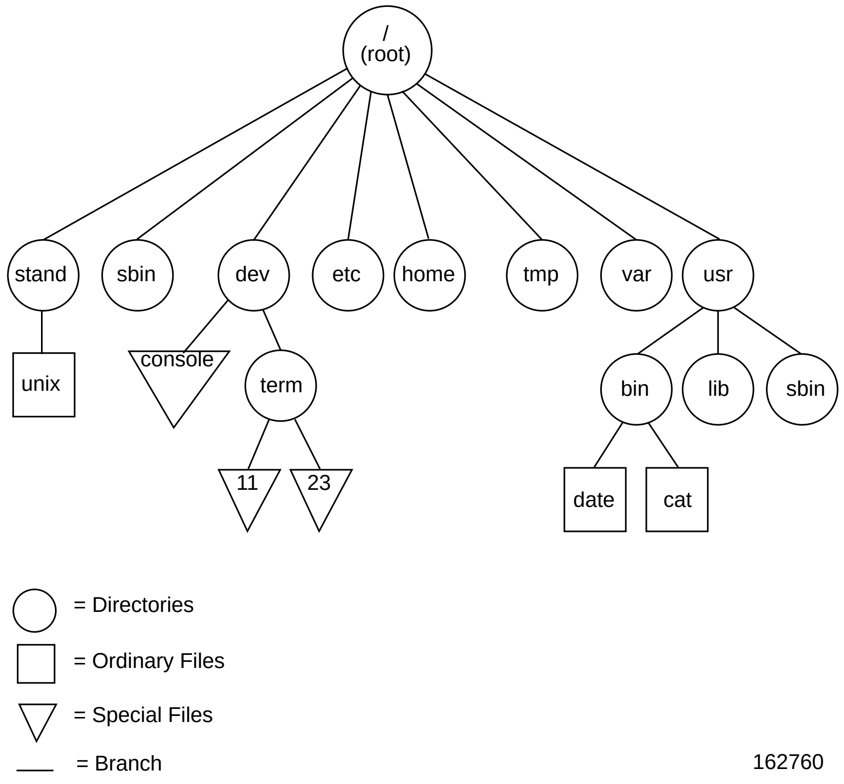


Figure 4-1. A Sample File System

Your Place in the File System

Whenever you interact with the UNIX system, you do so from a location in its file system structure. You are automatically placed at a specific point in the file system structure 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.

Home Directory

When you successfully complete the login procedure, you are placed at a specific point in the file system structure called your home directory (login directory). The login name assigned to you when your 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 `home`. Figure 4-2 shows the position of a system directory such as `home` in relation to the other important directories discussed in Chapter 2, “*What Is the UNIX System*”.

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

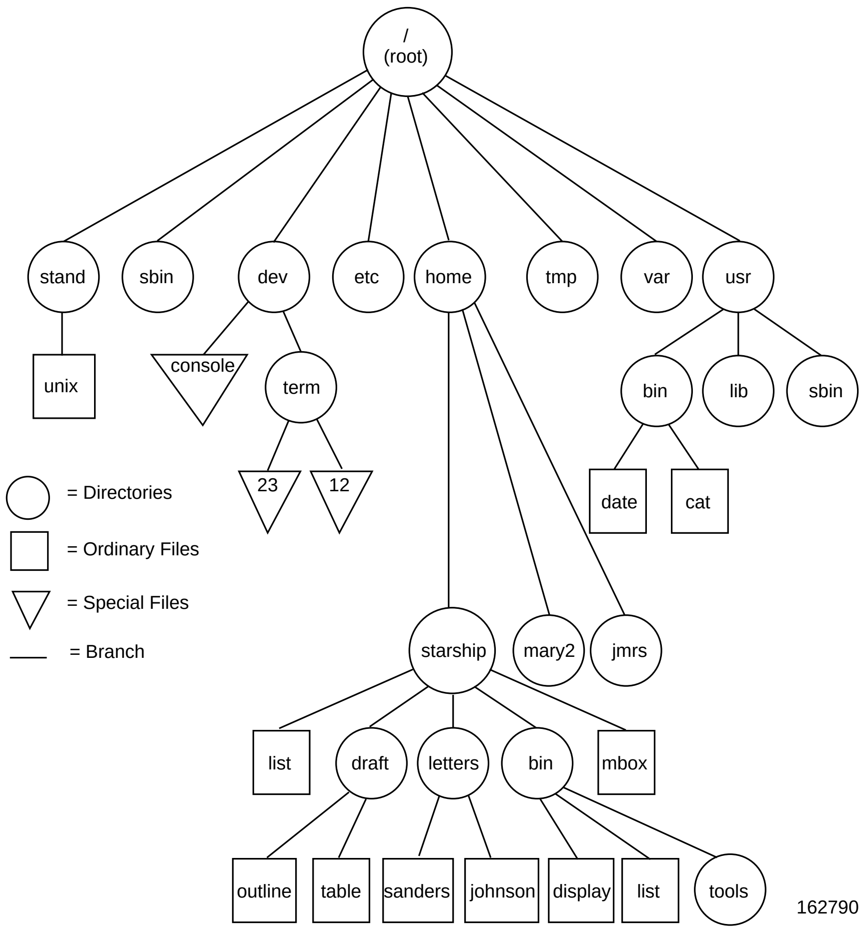


Figure 4-2. Directory of Home Directories

Current Directory

While 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 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>
/home/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 name **`/home/starship`** tells you that the root directory (shown by the leading **`/`** in the line) contains the directory `home` 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 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.

Table 4-1 provides a summary of the syntax and capabilities of the **`pwd`** command.

Table 4-1. Summary of the pwd Command

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

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 is to learn about the two types of pathnames: full and relative.

Full Pathnames

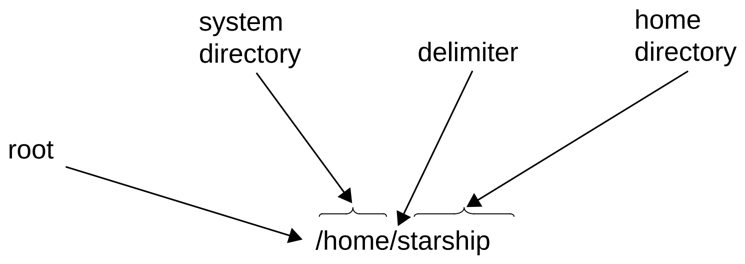
A full pathname (sometimes called an absolute pathname) gives directions that start in the root directory and lead you down through a unique sequence of directories to a particular directory or file. You can use a full pathname to reach any file or directory in the UNIX system in which you are working.

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

/home. You issue the **pwd** command and the system responds by printing the full pathname of your working directory: **/home/starship**.

Analyze the elements of this pathname using the diagram and key in Figure 4-3.



`/` (leading)	=	the slash that appears as the first character in the path-name is the root of the file system
`home`	=	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 `home` and `starship`
`starship`	=	current working directory

Figure 4-3. Pathname Elements

Now follow the dashed lines in Figure 4-4 to trace the full path to **/home/starship**.

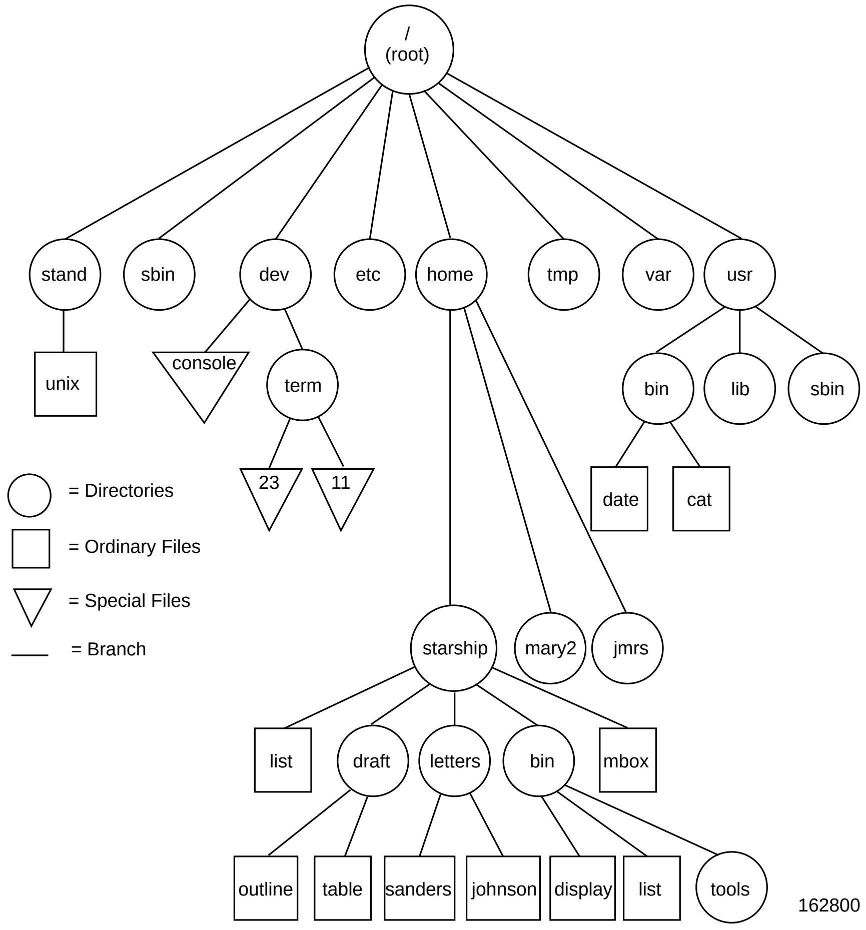


Figure 4-4. Full Pathname of the /home/starship Directory

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, suppose 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 4-5 traces the relative path from `starship` to `draft`.

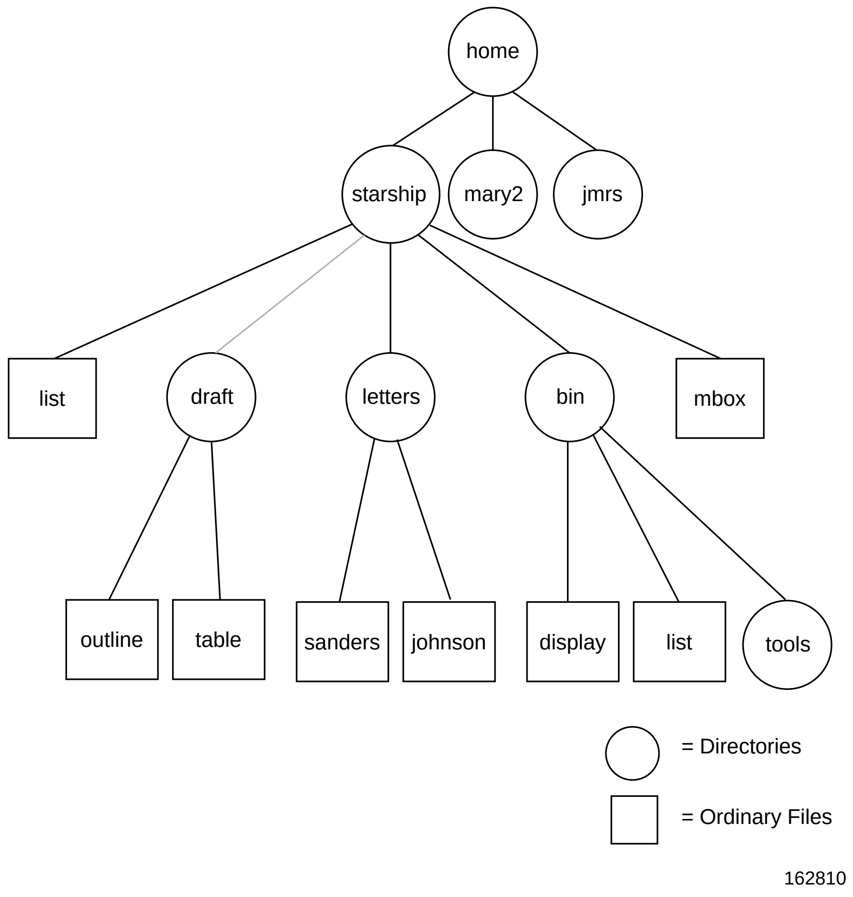


Figure 4-5. Relative Pathname of the draft Directory

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 4-6 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`; **`outline`** is a child of its parent, `draft`.

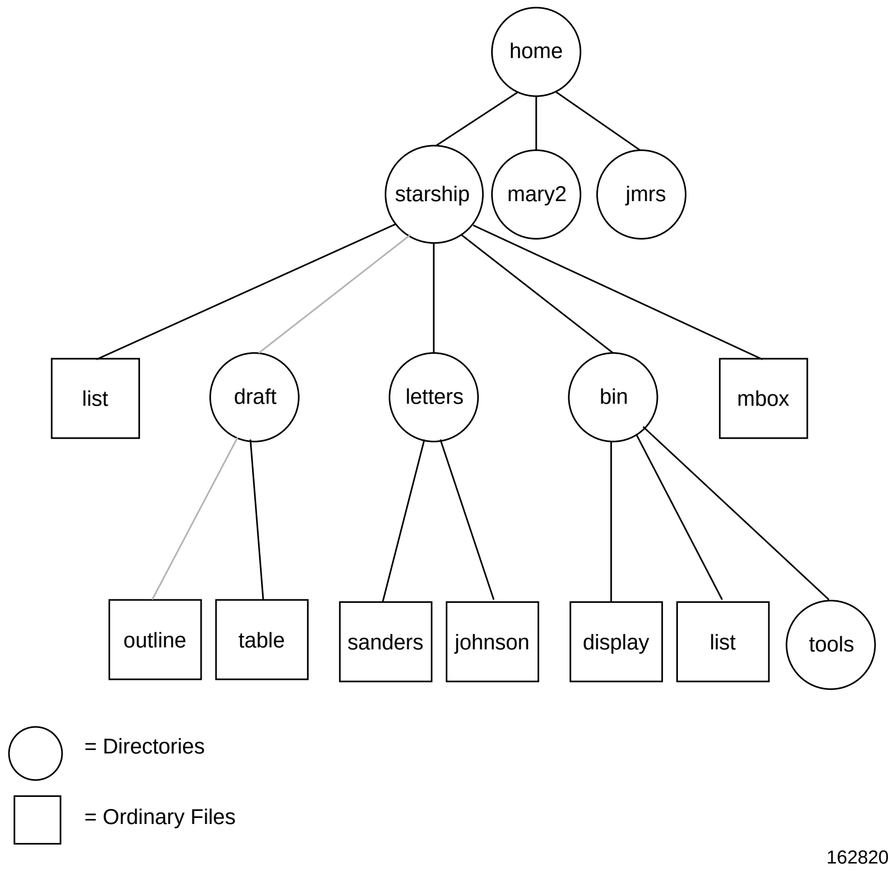


Figure 4-6. Relative Pathname from starship to outline

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 ascend the levels in the system structure or ascend and subsequently descend into other files and directories.

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

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.

You can always use a full pathname in place of a relative one.

Table 4-2 gives examples of full and relative pathnames.

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

Table 4-2. Example Pathnames

Pathname	Meaning
`/`	full pathname of the root directory.
`/usr/bin`	full pathname of the `bin` directory that belongs to the `usr` directory that belongs to `root` (contains most executable programs and utilities).
`/home/starship/bin/tools`	full pathname of the `tools` directory belonging to the `bin` directory that belongs to the `starship` directory belonging to `home` that belongs to root.
`bin/tools`	relative pathname to the file or directory `tools` in the directory `bin.` If the current directory is **`/usr`**, then the UNIX system searches for **`/usr/bin/tools`**, but if the current directory is `starship`, then the system searches the full path **`/home/starship/bin/tools`**.
`tools`	relative pathname of a file or directory `tools` in the current directory.

Directory and File Names

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

- 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.
- Upper case and lower case characters are distinct. 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 commands that enable you to examine the file system.

Security and Files

The security policy for the UNIX system with enhanced security installed requires that access to files be controlled by the Trusted Computing Base (TCB). The TCB keeps track of all files and makes all access control decisions accordingly.

The UNIX system uses the following kinds of files:

- An ordinary or regular file is a one-dimensional array of bytes. The UNIX system imposes no other constraint on the format of regular files; the kernel never changes regular file contents except as specified by user programs. This protects the contents of those files. Any structure or constraints on ordinary files are imposed by the programs that use the files.
- A special file (also called a device special file or device) represents an I/O device such as a terminal or disk, or a pseudo-device such as primary memory. Special files reside in the directory **`/dev`**. A device interface typically is partly conventional (most devices allow read, write, open, and close) and partly device-dependent (many devices have control operations specific to the device).
- A symbolic link allows a link between files that exist on different file systems. A symbolic link file is essentially a regular file with a unique type identifier; the data of the symbolic link file is the pathname of the file to which it is linked. See **`ln(1)`** in the online *Command Reference*.
- A directory organizes files, including other directories. The files in a running system form a hierarchy that begins at the root directory, which is denoted by a slash (`/`). Unlike regular files, user programs are not allowed to change the internal structure of directories; directories are maintained solely by the operating system.
- A pipe is a communication path set up by the kernel that can be used to pass data between processes. Users create pipes by using the **|** character to use the output from one program as the input to another, for example, `cat` *file* | sort.

As a user, you will need to use ordinary files and be aware of the directory structure of files in your day-to-day tasks. You may use symbolic links and pipes, and if you write any programs, may also make use of device special files.

Refer to Figure 4-1 for a sample file tree. Each file has an absolute or full pathname, which starts at root, ends with the file, and includes every directory on the path from the root to the file. For example, **`/usr/bin/date`** is the full pathname of the file **`date`**. **`/usr/bin`** is the full pathname of the directory `bin`.

A file can always be referenced by its full pathname.

Part of the information the TCB keeps about a process is its current working directory. Given a particular current working directory, each file below that directory has a relative pathname. A relative pathname starts with reference to the current directory, ends with the file, and includes every directory in between. For example, if the current working directory is **`/usr/bin`**, then **`date`** can be referenced by using the relative pathname **`date`**.

Remember that all files are assigned security levels by the TCB. Only processes that pass Mandatory and Discretionary access control checks are allowed to access any given file.

Organizing a Directory

This section introduces four commands that enable you to organize and use a directory structure:

`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

You should create subdirectories in your home directory according to a logical and meaningful scheme that will facilitate the retrieval of information from your files. If you put all files about one subject 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 (**`/home/starship`**):

```
$ mkdir draft <RETURN>
$
```

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

This example shows that other subdirectories were created in the home directory, such as `letters` and `bin`, in the same way.

```
$ mkdir letters <RETURN>
$ mkdir bin <RETURN>
$
```

All three subdirectories (`draft`, `letters`, and `bin`) could have been created 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 *"Directory and File Names."*

Table 4-3 summarizes the syntax and capabilities of the **`mkdir`** command.

Table 4-3. Summary of the mkdir Command

Command Recap **`mkdir`** - make a new directory		
command	*options*	*arguments*
`mkdir`	available	*directoryname(s)*
Description:	**`mkdir`** creates a new directory (subdirectory).	
Remarks:	The system returns a prompt (`$` by default) if the directory is successfully created.	

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 file system in Figure 4-2 once again.

Suppose you are logged in to the UNIX system as `starship` and you run the **`pwd`** command. The system will respond with the pathname **`/home/starship`**. To display the names of files and directories in this current directory, you then type **`ls`** and press the <RETURN> key. After this sequence, your terminal will read:

```
$ pwd <RETURN>
/home/starship
$ ls  <RETURN>
bin
draft
letters
list
mbox
$
```

The system responds by listing, in alphabetical order, the names of files and directories in the current directory `starship`. (If the first character of any of the file or directory names had been a number or an upper case 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 the `<RETURN>` key. 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 `home`:

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

You can get the same results by using the full pathname from root to `home`.

If you type **`ls /home`** and press the `<RETURN>` key, 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: No such file or directory
$
```

The system acknowledges the existence of **`outline`** by printing its name, and indicates 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 *"File Access and Manipulation"* later in this chapter.

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 online *Command Reference* for details about other options.

Listing All Files in a Directory

Some important file names in your home directory, such as **`.profile`** (pronounced dot-profile), begin with a period. (As you can see from this example, when a period is used as the first character of a file name it is pronounced dot.) When a file name begins with a dot, it is not included in the list of files reported by the **`ls`** command. If you want the **`ls`** to include these files, use the **`-a`** option on the command line.

For example, to list all the files in your current directory (`starship`), including those that begin with a **`.`** (dot), type **`ls -a`** and press the `<RETURN>` key.

```
$ ls -a <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 `*`), directories (with a `/`), and symbolic links (with an `@`). 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, suppose 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
$
```

`d =`	directory
`- =`	ordinary disk file
`l =`	symbolic link file
`b =`	block special file
`c =`	character special file

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 tells you the type of file as indicated above.

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

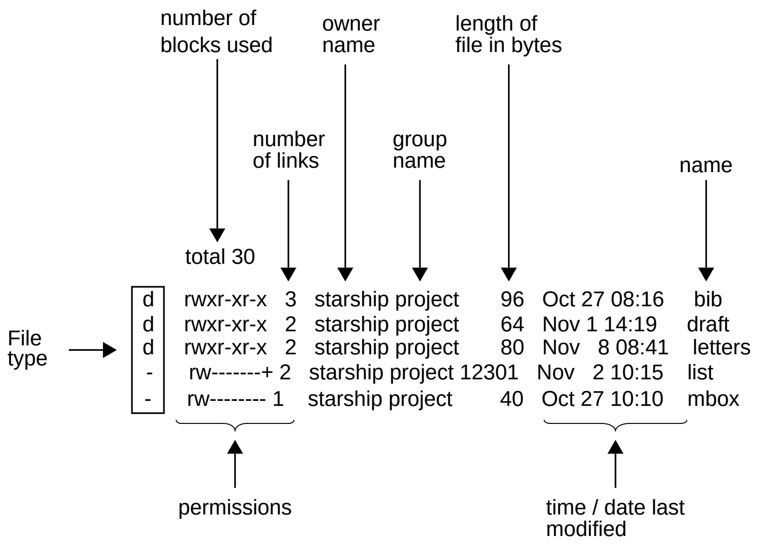
The next nine characters, which are either letters or hyphens, possibly followed by a plus sign (+), identify who has permission to read and use the file or directory. Permissions are discussed in the description of the **`chmod`** command under later in this chapter. The presence of a plus sign after the nine permission characters indicates additional permissions are granted and/or denied in the file's Access Control List (ACL). An ACL may further restrict access to the file beyond the restrictions specified by the permission characters. Permissions and ACLs are discussed in the section *“Protecting 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 the file was last modified is given next. Finally, the last column shows the name of the directory or file.

Figure 4-7 identifies each column in the rows of output from the **`ls -l`** command.



```
total 30
d  rwxr-xr-x   3  starship project     96  Oct 27 08:16   bib
d  rwxr-xr-x   2  starship project     64  Nov 1 14:19    draft
d  rwxr-xr-x   2  starship project     80  Nov  8 08:41   letters
-  rw-------+  2  starship project  12301  Nov  2 10:15   list
-  rw--------  1  starship project     40  Oct 27 10:10   mbox
```

Figure 4-7. Description of Output Produced by the ls-l Command

Two other options of the **`ls`** command, **`-z`** and **`-Z`**, allow you to display a file's Mandatory Access Control level, also called a security level. (These options are only available with the enhanced security package.)

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

Changing the Current Directory: The cd Command

When you first log in, you are placed in your home directory. As long as you do work in it, it is also your current working directory. By using the **`cd`** command (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.

Table 4-4. Summary of the ls Command

Command Recap **ls** - list contents of a directory		
command	*options*	*arguments*
ls	**-a**, **-l**, and others[1]	*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). **-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.	

1. See the **ls(1)** man page in the online *Command Reference* for all available options and an explanation of their capabilities.

For example, to move from the `starship` directory to its child directory `draft` (in the sample file system), type `cd draft` and press the <RETURN> key. (Here `draft` is the relative pathname to the desired directory.) When you get a prompt, you can verify your new location by typing **pwd** and pressing the <RETURN> key. Your terminal screen will look like this:

```
$ cd draft <RETURN>
$ pwd <RETURN>
/home/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. (See Chapter 6, the "*Line Editor (ed) Tutorial*" and Chapter 7, the "*Screen Editor (vi) Tutorial*" for tutorials on the **ed** and **vi** commands, respectively.)

It is not necessary to be in the `draft` directory to access files within it. You can access a file in any directory by specifying a full or relative pathname for it. For example, to **cat** the **sanders** file in the `letters` directory (**/home/starship/letters**) while you are in the `draft` directory (**/home/starship/draft**), specify the full pathname of **sanders** on the command line.

```
cat /home/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 **`/home/starship/letters`** on the command line, as follows:

```
cd /home/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`.

Table 4-5 summarizes the syntax and capabilities of the **`cd`** command.

Table 4-5. Summary of the cd Command

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

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 (the directory 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, suppose 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>
    UX:rmdir: ERROR:memos:Directory 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 *"File Access and Manipulation"* later in this chapter.

Once you have removed the contents of the memos directory, memos itself can be removed. First, 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.

Table 4-6 summarizes the syntax and capabilities of the **rmdir** command.

Table 4-6. Summary of the rmdir Command

Command Recap **rmdir** - remove a directory		
command	*options*	*arguments*
rmdir	available	*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.	

Protecting Files

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. Access to a file is controlled by two mechanisms: Discretionary Access Control (DAC) and Mandatory Access Control (MAC).

DAC allows you to grant or deny permission for other users to access your files. This feature is called discretionary access control because the owner has the discretion to grant or deny access. This chapter explains DAC, and how to use the DAC commands **`setacl`**, **`getacl`**, and **`chmod`** to control access to files you own.

NOTE

It is important to understand that a user who has been given permission to read a file is allowed to make copies of that file. Those copies will be owned (and access to them controlled) by that user.

MAC is controlled by the system (as configured by the System Administrator), and restricts a user's access to data based on the sensitivity and topics associated with the data and the user. The owner of a file has no control over the MAC restrictions on the file. Mandatory Access Controls are explained in Chapter 14, "*Managing Files Securely*" of this guide.

You must pass both sets of access checks to read or modify a file or to execute a program. Throughout the discussion of DAC, it is assumed that the user has MAC access to the files or directories involved.

Overview

If you own a file, DAC allows you to 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. When you grant execute permissions for a directory, you allow the specified users to change directory to it. Discretionary access to any file, directory, or device is controlled by the owner or an administrator with the appropriate privilege.

DAC permissions for a file are stored in two kinds of file attributes. A file's Access Control List (ACL) contains one-line entries naming specific users and groups and indicating what access is to be granted to each. A file's permission bits represent in a shorthand fashion ACL entries for the file's owner and group, and for "other" (anyone who is not specifically granted or denied access by other ACL entries).

Using the **`setacl`** command you can selectively grant or deny any user or group of users access to your files. The **`chmod`** command provides a limited subset of the functions of the **`setacl`** command, manipulating only the ACL entries represented in the permission bits. The **`chgrp`** and **`chown`** commands also affect the DAC information on a file.

The following sections explain types of file access, access control lists, and permission bits in more detail. This is followed by detailed discussions of how to display ACLs and permission bits using the **`getacl`** and **`ls`** commands, and how to control access to your files using **`setacl`** and **`chmod`**.

Types of Access

Three specific types of access to a file are controlled by the owner of a file or directory:

read	For a file, read access allows a user to see the contents of the file. For a directory, read access allows a user to access files within the directory.
write	For a file, write access allows a user to change the contents of the file. For a directory, write access allows a user to create new files, append to existing files, and remove existing files.
execute	If a file is a program or a shell script, execute access allows a user to execute it. For a directory, execute permission allows a user to make the directory the current directory and to obtain information on the attributes of files within the directory.

When displayed using the **`ls -l`** command, access permissions are written out as a sequence of nine characters, divided logically into three sets of three characters each. There is one set for the permissions to be granted to the file's owner, members of the file owner's group, and all others, displayed in that order.

Each set contains three characters, with `r` standing for read permission, `w` standing for write permission, and `x` standing for execute permission. The characters are displayed in this order for each set, with a dash (`-`) indicating that a particular permission is not granted. Refer to Screen 4-3 for an example of displaying permissions.

Because of the way permissions map onto individual bits in the operating system code, a set of permissions are often referred to as permission bits or mode bits.

The next section explains how to interpret permission bits as displayed by **`ls -l`**, and how Access Control Lists can modify the permissions granted to particular users and groups.

Access Control Lists and Permission Bits

Permission bits provide an easy-to-use mechanism for controlling access to a file. They are limited, however, because there are only three sets of permissions.

The permission bits appear in three sets of three characters, in the order `owner, group,` and `other.` For example, in Screen 4-3, only the owner of the file has read and write access to the file **`mbox`**. No one in the group `project`, nor anyone else, has access to the file. Permissions that would grant yourself read and write access to the file, members of your group read-only access, and no access at all to others would appear `rw-r-----.` The `group` permissions would apply to all users in the group `project`, the `other` permissions to everyone else on the system that is not in the group `project`.

The permission bits cannot grant and deny access to specific other users. There is no way to give another user who is not in the file's owning group access to the file without giving access to all users on the system. In order to grant and deny access to specific users, you need to add entries to the file's Access Control List (ACL).

Every file on the system has an ACL. An ACL consists of a series of one-line entries naming specific users or groups and indicating exactly what access is to be granted to each.

There are always at least four entries in an ACL, a `user` entry, a `group` entry, a `class` entry, and an `other` entry. When an ACL contains only four entries, the permissions it grants are exactly the same as the permissions represented by the permission bits.

While having such an ACL (we will call it a minimal ACL) provides no greater functionality than the permission bits alone, we will start by describing a minimal ACL, and augment it with additional entries to show how the DAC mechanism works.

Minimal ACL

The first entry in a minimal ACL indicates the permissions that the owner of the file gets, and maps directly to the `owner` permission bits. Because it applies to the owner of the file, no indication of the user's name is needed. An ACL entry that grants read and write access to the file's owner would look like this:

```
user::rw-
```

The second and third entries in a minimal ACL specify the permission granted to members of the file's owning group; the permissions specified in these entries are exactly equal. For example, ACL entries granting read-only access to the file's owning group would look like this:

```
group::r--
class::r--
```

The fourth and last entry in a minimal ACL is a catch-all entry that specifies the permissions for anyone who isn't granted or denied permission by any other entry. An `other` entry that denies access to all users not the owner of the file nor in the file's owning group would look like this:

```
other:---
```

The minimal ACL described above would look like this in its entirety:

```
user::rw-
group::r--
class::r--
other:---
```

The permission bits displayed by **`ls -l`** for this file would look like this:

```
rw-r-----
```

In the case of a minimal ACL, there is a clear correspondence between the ACL entries and the permission bits.

The next section describes how additional ACL entries affect file access and the interpretation of the permission bits.

Additional ACL Entries

If you want to specifically grant and/or deny access to specific users and/or groups on the system, you can add more `user` and `group` entries to the four minimal entries described in the previous section.

Additional `user` entries grant and deny access to specific user IDs on your system. For example, the following entry in the ACL of a file grants read, write, and execute access to a user logged in as `archer:`

```
user:archer:rwx
```

Similarly, additional `group` entries grant and deny access to specific group IDs on your system. For example, an ACL with the following entry would deny access to a user in the group `spies:`

```
group:spies:---
```

In an ACL that contains more than one `group` entry and/or more than one `user` entry, the `class` entry specifies the maximum permissions that can be granted by any of the additional `user` and `group` entries. If a particular permission is not granted in the `class` entry, it cannot be granted by any ACL entries (except for the first `user` (owner) entry and the `other` entry). Any permission can be denied to a particular user or group. The `class` entry functions as an upper bound for file permissions.

When an ACL contains more than one `group` and/or `user` entry, the collection of additional `user` and `group` entries are referred to as the `group class` entries, since the effective permission granted by any of these additional entries is limited by the `class` entry.

If there are additional entries in the ACL, the `class` ACL entry will no longer necessarily equal the value of the permissions for the owning group reported by **`ls -l`**. This feature is useful, because it means that the **`chmod`** command can usefully affect the permissions of a file that has additional ACL entries.

For example, by changing the permission bits of a file to `rwx------`, the `class` entry in the ACL is set to `---`. This means that any additional `group` entries in the ACL cannot grant any access to the file. If the permission bits were set to `rwxr-----`, the `class` ACL entry would be `r--`, and any `group` entries would be able to grant read access, but not write or execute access.

The `class` entry does not limit the access that can be granted by the first `user` (owner) entry or the `other` entry.

Displaying a File's Permission Bits and ACL

You can display the permission bits of a file with the **`-l`** option of the **`ls`** command, or display the complete ACL with the **`getacl`** command.

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:

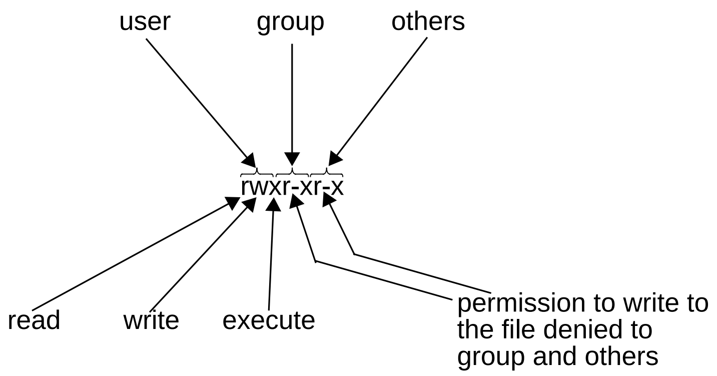
```
$ ls -l  <RETURN>
total 35
-rwxr-xr-x   1 starship      project        9346  Nov 1  08:06  display
-rw-r--r--+  1 starship      project        6428  Dec 2  10:24  list
drwx--x--x   2 starship      project          32  Nov 8  15:32  tools
$
```

Permissions for the **`display`** and **`list`** files and the `tools` directory are shown on the left of the screen under the line `total 35`.

The initial character describes the file type; for example, a dash (`-`) indicates a regular file, and a `d` a directory. The next nine characters are the three sets of permission bits, as described previously. The first set refers to permissions for the owner, the second set to the maximum group class permissions, 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.

At the end of these characters you may also see a plus sign indicating the presence of additional ACL entries. This indicates that additional permissions, beyond those indicated by the three sets of three bits, have been granted or denied by additional ACL entries. In the example, only the file named **`list`** has additional ACL entries.

The following diagram summarizes the breakdown of permissions for the file named **`display`**.



As shown, the owner has `r`, `w`, and `x` permissions and members of the group and other system users have `r` and `x` permissions.

There are several exceptions to this notation system. Occasionally other letters such as `s` or `l` may appear in the permission bits, instead of an `r`, `w` or `x`.

The letter `s` means that the file will execute with an effective user or group ID of the owning user or group. It appears where you normally see an `x` (or -) for the user or group (the first and second sets of permissions).

The letter `l` indicates that locking will occur when the file is accessed. It does not mean that the file has been locked. See the **`ls(1)`** and **`chmod(1)`** manual pages in the online *Command Reference* for more information on variation of the standard `rwx` permissions.

To view the ACL on a file, use the **`getacl`** command, which prints a file's DAC information to standard output. In addition to the ACL, it prints the file name and information on the owning user and group. An example of using the **`getacl`** command on a file with only the four basic ACL entries is shown in Screen 4-1.

```
$ getacl sago <RETURN>
# file: sago
# owner: fletcher
# group: tourney
user::rw-
group::rw-
class:rw-
other:---
```

Screen 4-1. A simple ACL

User `fletcher` has read and write permissions, as does anyone in the group `tourney.` No one else has any access to the file.

Since the file in the above example has only the basic four ACL entries, you can get the same information using either the **`-l`** option of **`ls`** or the **`getacl`** command. For a file with additional ACL entries, only the **`getacl`** command displays complete DAC information.

Table 4-7 summarizes the syntax and capabilities of the **`getacl`** command.

Table 4-7. Summary of the getacl Command

Command Recap **`getacl`** - display Access Control Lists for files (and directories)		
command	*options*	*arguments*
`getacl`	available[1]	*file(s) or directories*
Description:	**`getacl`** displays the Access Control Lists of files and directories.	
Remarks:	The **`ls -l`** command displays the owner, group, and basic ACL entries in a shorter format.	

1. See the **`getacl(1)`** page in the online *Command Reference* for all available options and an explanation of their capabilities.

Changing the Access Control List of a File

If you are `fletcher` (the owner of the file), you can provide read access to an additional user by adding an entry to the ACL naming that user and specifying read access. For example, the following command gives user `archer` read-only access to the file:

```
setacl -r -m u:archer:r-- sago
```

The **`-m`** option indicates that you are adding an entry to the ACL. The **`-r`** automatically recalculates the `class` entry for you, so that any permissions that you specify will actually be granted.

You can add group-specific entries in just the same way. For example, to grant read and write access to everyone in the group `judges,` type the following:

```
setacl -r -m g:judges:rw- sago
```

The **`-m`** (modify) option can be used to change an existing entry as well as add a new one. If an entry already exists for the specified user or group, the permissions for that entry are set to the values specified on the command line.

There is also a **`-d`** option, to delete an entry. When the **`-d`** option is specified, you do not specify any permissions in the ACL entry. For example, the following command deletes an entry for the group `judges:`

```
setacl -r -d g:judges
```

Any number of entries may be added using the **`-m`** option.

If you are adding or changing several entries, you will probably want to use a different procedure. You can save the ACL to a file, edit it, adding, changing, or deleting entries to produce whatever ACL you want, and then apply this new ACL to the file. For example, you could save the ACL to a file with this command:

```
getacl sago > sago.acl
```

Then you could edit it so that it appeared as in Screen 4-2.

```
# file: sago
# owner: fletcher
# group: tourney
user::rw-
user:archer:rw-
user:bowman:rw-
user:hunter:---
group::rw-
group:audit:r--
group:fencers:---
group:judges:rw-
class:rw-
other:r--
```

Screen 4-2. A Complex ACL

This ACL can now be applied to the file by using the **`-f`** option of the **`setacl`** command as follows:

```
    setacl -r -f sago.acl sago
```

In this example several changes have been made. In particular, the other permissions, which apply to everyone not mentioned in the ACL now grant read access. While before the ACL entries only granted access to people, now they are also used to deny access as well. Note specifically the entries for user `hunter` and group `fencers.`

Table 4-8 summarizes the syntax and capabilities of the **`setacl`** command.

Table 4-8. Summary of the setacl Command

Command Recap **`setacl`** - Modify the Access Control Lists for files (and directories)		
command	*instruction*	*arguments*
`setacl`	**`-r -m`** *type:id:permission*	*filename(s)*
		directoryname(s)
`setacl`	**`-r -d`** *type:id:permission*	*filename(s)*
		directoryname(s)
`setacl`	**`-r -f`** *aclfile*	*filename(s)*
		directoryname(s)
Description:	**`setacl`** changes the Access Control Lists of files and directories.	
Remarks:	Use the **`-m`** option to add an ACL entry, or the **`-d`** option to delete an entry. Use the **`-f`** entry to specify a file containing a complete ACL to be assigned to files or directories.	

The following section presents cautions on interpreting complex ACLs.

Determining Access

In a complex ACL, it sometimes requires a moment's thought to see what access any particular user would receive. The user-specific entries are all checked before the group-specific entries; if a user appears in a user-specific entry, group-specific entries have no effect on the permissions granted.

If a user is granted permission by any group-specific entry, or any combination of group-specific entries, other group-specific entries cannot take that permission away. Only if no user- or group- specific ACL entries apply to a user are the permissions of the `other` entry used.

In the ACL shown in Screen 4-2, access is determined as follows:

- User `fletcher,` as file owner, gets read and write access to the file.
- Users `archer` and `bowman` get read and write access to the file.
- User `hunter` gets no access to the file. This is true even if `hunter` is in group `judges,` `audit,` or `tourney.`
- Members of group `tourney` get read and write access to the file.
- Members of group `judges` get read and write access to the file.
- Members of group `audit` get read only access to the file, but others in group `judges` or `tourney` still get read and write access.
- Members of group `fencers` get no access to the file, but members of the groups `judges,` `audit,` or `tourney,` and users `archer,` `fletcher,` and `bowman` are not affected by this entry.
- Other users of the system get read-only access to the file.

Remember that the `class` entry limits the permissions that can be granted by the group class entries. If a particular permission is not granted in the `class` entry, it cannot be granted by the ACL, except through the first (owner) `user` entry owner or the `other` entry. When an additional `user` or `group` entry specifies greater permission than the limit specified by the `class` entry, the **`getacl`** command indicates this with a line in the form

```
#effective:---
```

as in Screen 4-3.

```
# file: sago
# owner: fletcher
# group: tourney
user::rw-
user:archer:rw-        #effective:---
user:bowman:rw-        #effective:---
user:hunter:---
group::rw-             #effective:---
group:audit:r--        #effective:---
group:fencers:---
group:judges:rw-       #effective:---
class:---
other:---
```

Screen 4-3. Effect of class entry on an ACL

In the example, the permissions specified for user `archer` (`rw-`) is greater than the permissions specified by the `class` entry (`---`); hence, the effective permissions granted to user `archer` is `---`.

The additional `user` and `group` ACL entries remain; however, they grant no access because of the limiting affect of the `class` entry.

It is always a good idea to display an ACL with **`getacl`** after you alter it, to make sure that the ACL grants and denies access as you intend. This is particularly true when you use the **`-r`** option to **`setacl`** to recalculate the `class` entry.

For example, assume the following ACL:

```
user::rw-
user:archer:rw-
group::rw-
group:audit:r--
class:---
other:r--
```

Access is denied to user `archer` and group `audit` by the `class` entry, despite the access specified in their entries. If you execute the following command:

```
setacl -r -m g:steno:r file
```

The resulting ACL looks like this:

```
user::rw-
user:archer:rw-
group::rw-
group:audit:r--
group:steno:r--
class:r--
other:r--
```

The recalculated `class` entry now gives read access to both user `archer` and group `audit`, as well as the group you just added, `steno`. If you still want to deny access to `audit` and `archer,` you should change their ACL entries to `---` with **`setacl.`**

Assigning Permissions: The chmod Command

The **`chmod`** command provides an alternative method to modify the user, group class, or other permissions on a file or files. To assign these types of permissions with the **`chmod`** command 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.

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)

u = user

g = group

o = others

+ or - instruction that grants (+) or denies (-) permission

permission	any combination of three authorizations (r, w, and x)
	r = read
	w = write
	x = execute
file(s)	file (or directory) name(s) listed; assumed to be branches from your current directory, unless you use full pathnames.

NOTE

The **chmod** command will not work if you type 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 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. Permission to write into the file **`display`** has been denied to members of your group and other system users, but not 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
$
```

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.)

In the octal method, the first digit represents the `owner` permissions, the second the `class` entry, (which is equal to the owning group's permissions), and the third `other` permissions. The digit is constructed by treating the three types of permissions as bits, with read having a value of 4, write having a value of 2, and execute having a value of 1. A digits value is calculated by summing the values of permissions which are granted. Thus, `rwxrwxrwx` would be indicated as `777` and `rw-r-----` would be indicated as `640.` To learn how to use the octal method, see the **`chmod(1)`** entry in the online *Command Reference.*

Table 4-9 summarizes the syntax and capabilities of the **`chmod`** command.

Table 4-9. Summary of the chmod Command

Command Recap **chmod** - change permission modes for files (and directories)		
command	*instruction*	*arguments*

chmod	*who + - permission*	*filename(s)*
		directoryname(s)

Description:	**chmod** gives (+) or removes (-) permission to read, write, and execute files for three categories of system users: `user` (you), `group` (members of your group), and `other` (all other users able to access the system on which you are working).	
Remarks:	The instruction set can be represented in either octal or symbolic terms.	

Permissions and Directories

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

Consider the impact on various system users of changing permissions for directories. For example, suppose 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.

Default Access Control Lists

Often, you will want all the files created in a directory to have certain ACL entries. For example, you might want to allow another person to write to any file in a directory of yours where the two of you are working on something together.

You can put an ACL entry granting the desired access on every file in the directory, but every time you create a new file you will have to add that entry again. Using default ACL entries, you can get the system to do this for you automatically every time a file is created.

A default ACL entry looks like this:

```
default:user:archer:rw-
```

It can be placed only on a directory, never on an ordinary file. It never has any influence on what access is granted to a user for the directory it is placed on. All it does is cause the specified entry to be included in the ACL of any file created in the directory.

If the newly created file is a directory, the default ACL entries have two effects. First, the corresponding non-default ACL entries are created, so that the desired permissions are granted and denied for the directory, just as for any file created in the directory. Second, the default entries themselves are copied, so that new sub-directory has the same default ACL as the parent directory.

For example, if you want any files created in the directory `poentkarto` to be readable by certain users, you could create the appropriate default entries as shown in Screen 4-4.

```
# file: poentkarto
# owner: fletcher
# group: tourney
user::rw-
user:archer:rw-
user:bowman:rw-
user:hunter:---
group::rw-
group:judges:rw-
class:rw-
other:---
default:user:archer:r--
default:user:bowman:r--
default:group:judges:r--
```

Screen 4-4. An ACL with default entries

With these entries in place, any new file created in the directory `poentkarto` would have an ACL like that shown in Screen 4-5.

```
# file: poentaro1
# owner: fletcher
# group: tourney
user::rw-
user:archer:r--
user:bowman:r--
group::rw-
group:judges:r--
class:rw-
other:---
```

Screen 4-5. Effect of default entries on a file

If the newly created file is a directory, the same ACL entries are generated, but in addition the default entries themselves are also placed in the ACL, as shown in Screen 4-6.

```
# file: subpoento
# owner: fletcher
# group: tourney
user::rw-
user:archer:r--
user:bowman:r--
group::rw-
group:judges:r--
class:rw-
other:---
default:user:archer:r--
default:user:bowman:r--
default:group:judges:r--
```

Screen 4-6. Effect of default entries on a directory

There is another mechanism, called `umask` for controlling the DAC permissions of newly created files. Unlike a default ACL entry, the `umask` value affects the permissions of any file created, regardless of the directory it is created in. The `umask` consists of three octal digits, much like the permission bits. Rather than granting permission, they act as a mask, indicating permissions that are not to be granted. Thus, a `umask` value of `022` indicates that write permission should not be granted to anyone except the owner of any newly created files.

The system administrator defines a default `umask` that applies to files and directories created by any user. You can override this default `umask` by modifying your environment (see Chapter 9, "*Shell Tutorial*" in this guide, and the **`umask(1)`** manpage in the online *Command Reference* for details).

It is important to understand that many commands specifically modify the permissions of files that they create. For example, if you copy a file with the **`cp`** command, the command will also copy the permissions. In this case, neither the default ACL entries nor the `umask` value will have any effect on the permissions of the new file.

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.

File Access and Manipulation

This section discusses commands that are necessary for accessing and using the files in the directory structure. Table 4-10 lists basic commands for using files; other, more advanced file manipulation commands are found further on in this section.

Each command is discussed in detail and summarized in a table that you can easily reference later. These tables will enable you to review the syntax and capabilities of these commands at a glance.

Table 4-10. Basic Commands for Using Files

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
`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 (bytes) in a file
`chmod`	changes access permissions for a file (or a directory)
`setacl`	sets the access control lists for a file (or a directory)

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

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 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 directs the formatted output to a printer (see the **`lp`** command in Chapter 8, the “*LP Print Service Tutorial*” chapter).

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

Concatenating and Printing the Contents of a File: The cat Command

The **`cat`** command displays the contents of a file or files. For example, suppose 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, 1994

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

Dear Mr. Johnson:

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

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

Yours truly,

John Howe
$
```

To display the contents of two (or more) files, type the names of the files you want to see on the command line. For example, to display the contents of the files **`johnson`** and **`sanders`**, 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, 1994

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, 1994

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 section that discusses input and output redirection in Chapter 9, the "*Shell Tutorial*".

Table 4-11 summarizes the syntax and capabilities 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 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. Table 4-12 summarizes some of the available instructions.

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 twenty-four lines, one page is defined as twenty-three lines of text and a line containing a colon.

If a file is less than twenty-three lines long, its page size will be the number of lines in the file plus one (for the colon).

Table 4-11. Summary of the cat Command

Command Recap **`cat`** - concatenate and print a file's contents		
command	*options*	*arguments*
`cat`	available[1]	*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. To display the contents of a directory, use the **`ls`** command.	

1. See the **`cat(1)`** page in the online *Command Reference* for all available options and an explanation of their capabilities.

Table 4-12. Summary of Commands Used with pg

Command[1]	Function
`h`	help; display list of available **pg**[2] commands
`q` or `Q`	quit **pg** perusal mode
<RETURN>	display next page of text
`l`	display next line of text
`d` or `^d`	display additional half page of text
`.` or `^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

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

2. See the online *Command Reference* for a detailed explanation of all available **pg** commands.

To look at 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`** from the directory `draft` 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 online *Command Reference).*

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

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

To specify a terminal type, assign the code for your terminal to the `TERM` variable in your **.profile** file. (For more information about `TERM` and **.profile**, see the "*Shell Tutorial*" chapter; for instructions on setting the `TERM` variable, see the "Summary of Shell Command Language" appendix.)

Table 4-13 summarizes the syntax and capabilities of the **pg** command.

Table 4-13. Summary of the pg Command

Command Recap **pg** - display a file's contents in chunks or pages		
command	*options*	*arguments*
pg	available[1]	*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.	

1. See the **pg(1)** page in the online *Command Reference* for all available options and an explanation of their capabilities.

Printing Files: The pr Command

The **pr** command is used to format and print the contents of a file. It supplies titles and headings, paginates, and prints a file on your terminal screen in any of various page lengths and widths.

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 Chapter 8, the “LP Print Service Tutorial”). You can also direct the output of **pr** to a different file (see Chapter 9, the “*Shell Tutorial*” chapter).

If you do not specify any of the available options, the **pr** command produces output in a single column that contains sixty-six 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. Check with your system administrator to see if this software is available to you.)

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.

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 1994 johnson Page 1


March 5, 1994

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 blank lines after the last line in the file represent the remaining lines (all blank in this case) that **pr** adds to the output so each page contains a total of sixty-six lines. If you are working on a video display terminal, which allows you to view twenty-four lines at a time, the entire sixty-six lines of the formatted file will be printed rapidly without pause. This means that the first forty-two 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.

If the file you are examining is particularly long, even this ability may not be sufficient to allow you to read the file. In these 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.

Table 4-14 summarizes the syntax and capabilities of the **pr** command.

Making a Duplicate Copy of a File: 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.

Table 4-14. Summary of the pr Command

Command Recap **pr** - print formatted contents of a file		
command	*options*	*arguments*
pr	available[1]	*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 sixty-six 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.	

1. See the **pr** page in the online *Command Reference* for all available options and an explanation of their capabilities.

To copy the file named **outline** to a file named **new.outline** in the `draft` 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.

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 /home/starship/letters/outline <RETURN>` (full pathname specified)
- `cp outline ../letters/outline <RETURN>` (relative pathname specified)

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. You could also use either of these command lines:

- `cp outline /home/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 /home/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 *"Directory and File Names"* in this chapter.

Table 4-15 summarizes the syntax and capabilities of the **`cp`** command.

Table 4-15. Summary of the cp Command

Command Recap **`cp`** - make a copy of a file		
command	*options*	*arguments*
		file1 file2
`cp`	available	*file(s) directory*
Description:	The **`cp`** command 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. If the arguments to the **`cp`** command are *file(s)* and *directory*, the **`cp`** command copies the *file(s)* into *directory.* You cannot copy directories with the **`cp`** command.	

Moving and Renaming a File: The mv Command

The **`mv`** (short for move) command 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, suppose you want to move the file **`table`** from the current directory named `draft` (whose full pathname is **`/home/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 **`/home/starship/letters`**), you can use any one of several command lines, including the following:

```
mv table /home/starship/letters <RETURN>
mv table /home/starship/letters/table <RETURN>
mv table ../letters <RETURN>
mv table ../letters/table <RETURN>
mv /home/starship/draft/table /home/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 /home/starship/letters/table2 <RETURN>
mv table ../letters/table2 <RETURN>
mv /home/starship/draft/table2 /home/starship/letters/table2 <RETURN>
```

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

Table 4-16 summarizes the syntax and capabilities 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 **`rm`** command (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.

Table 4-16. Summary of the mv Command

Command Recap **mv** - move or rename files		
command	*options*	*arguments*
		file1 file2
mv	available	*file(s) directory*
Description:	The **mv** command 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. If the arguments to the command are *file(s)* and *directory,* **mv** moves *file(s)* to *directory.*	

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

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

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

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

Table 4-17 summarizes the syntax and capabilities of the **rm** command.

Counting Lines, Words, and Characters 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. Character counts are provided in bytes. If you name more than one file, the **wc** command counts the number of lines, words, and characters in each specified file and then totals the counts. In addition, you can direct the **wc** command 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:

Table 4-17. Summary of the rm Command

Command Recap **rm** - remove a file		
command	*options*	*arguments*
rm	available[1]	*file(s)*
Description:	The **rm** command allows you to remove one or more files.	
Remarks:	Files specified as arguments to the **rm** command are removed permanently.	

1. See the **rm (1)** page in the online *Command Reference* for all available options and an explanation of their capabilities.

```
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 (bytes) 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 twenty-four lines, sixty-six 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** command 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 twenty-four lines, sixty-six words, and 406 characters. The second line reports twenty-eight lines, ninety-two words, and 559 characters in the **sanders** file. The last line shows that these two files together have a total of fifty-two 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:

wc **-l** *file1* <RETURN>	(line count)
wc **-w** *file1* <RETURN>	(word count)
wc **-c** *file1* <RETURN>	(character/byte 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 (bytes), respectively, in the file.

Table 4-18 summarizes the syntax and capabilities of the **wc** command.

See the *System Administration* for details about setting the user or group ID.) The letter `l` indicates that locking will occur when the file is accessed. It does not mean that the file has been locked.

Changing Existing File 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>

Table 4-18. Summary of the wc Command

Command Recap **`wc`** - count lines, words, and characters (bytes) 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. Character counts are provided in bytes.	
Options:	**`-l`** counts the number of lines in the specified file(s)	
	`-w` counts the number of words in the specified file(s)	
	`-c` counts the number of characters (bytes) in the specified file(s)	
Remarks:	When a file name is specified in the command line, it is printed with the count(s) requested.	

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`)
	`u` = user
	`g` = group
	`o` = others
`+` or `-`	instruction that grants (`+`) or denies (`-`) permission
permission	any combination of three authorizations (`r`, `w`, and `x`)
	`r` = read
	`w` = write
	`x` = execute
file(s)	file (or directory) name(s) listed; assumed to be branches from your current directory, unless you use full pathnames

NOTE

The **`chmod`** command will not work if you type 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, but 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
$
```

Setting Directory Permissions

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.

Consider the impact on various system users of changing permissions for directories. For example, suppose 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.

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)`** entry in the online *Command Reference.*

Table 4-19 summarizes the syntax and capabilities of the **`chmod`** command.

Table 4-19. Summary of the chmod Command

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

File Ownership: The chown, id, and groups Commands

Several other commands are useful when manipulating files. These include the **`chown,`** **`id,`** and **`groups`** commands. If you are the owner of a file, your login name is located in the *owner* category. The **`chown`** command allows you, the owner of a file, to change your owner ID to someone else's ID for that file. For example, if you type:

```
ls -l display <RETURN>
```

the following information will appear on the screen:

```
-r-xr-xr-x 1 owner   group   9346 Nov 1 08:06 display
```

In order to change the owner ID from yours to, for example, Sara's, whose login name is sara, you would type:

```
chown sara display <RETURN>
```

If you type:

```
ls -l display <RETURN>
```

the message on the screen will read:

```
-r-xr-xr-x 1 sara   group   9346 Nov 1 08:06 display
```

If you use the **chown** command and an error message is displayed across the screen, this would indicate that your system administrator has restricted this option when the system was initially set up.

If you type:

```
id <RETURN>
```

the system will display the user's ID (UID) and your effective group ID (GID).

Depending on the initial setup of the system, you may belong to more than one group. In order to find out which groups you are a member of type:

```
groups <RETURN>
```

A list of all groups to which you have membership will appear on the screen. You have access to files whose group ID matches one from your supplementary group list.

Advanced File Manipulation: The diff, grep, and sort Commands

As you become familiar with these commands, your need for more sophisticated information processing techniques when working with files may increase. This section introduces three more commands that begin providing 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 online *Command Reference.*

Identifying Differences between Files: 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 using **ed** (line editor) commands. (See Chapter 6, the "*Line Editor (ed) Tutorial*" chapter 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, suppose 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 (using a text editor such as **ed** or **vi),** the **johnson** file will be identical to the **sanders** file. Remember, the **diff** command identifies differences between specified files. If you want to make an identical copy of a file, use the **cp** command.

Table 4-20 summarizes the syntax and capabilities 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 globally search for a

Table 4-20. Summary of the diff Command

Command Recap **diff** - finds differences between two files		
command	*options*	*arguments*
diff	available[1]	*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).	

1. See the **diff(1)** page in the online *Command Reference* for all available options and an explanation of their capabilities

`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. (For an explanation of the special meaning for these and other characters see Chapter 9, the "*Shell Tutorial*" chapter.) For example, suppose 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.
$
```

The output 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 online *Command Reference* for further information about these commands.

Table 4-21 summarizes the syntax and capabilities of the **`grep`** command.

Table 4-21. Summary of the grep Command

Command Recap **`grep`** - searches a file for a pattern		
command	*options*	*arguments*
`grep`	available[1]	*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.	

1. See the **`grep (1)`** page in the online *Command Reference* for all available options and an explanation of their capabilities.

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 upper case letters are listed before lines beginning with lower case letters.
- Lines beginning with symbols such as `*`, `%`, or `@`, are sorted on the basis of the symbol's ASCII representation.

For example, suppose 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 online *Command Reference* for a full description of available options.

Table 4-22 summarizes the syntax and capabilities of the **`sort`** command.

Table 4-22. Summary of the sort Command

Command Recap **`sort`** - sorts and merges files		
command	*options*	*arguments*
`sort`	available[1]	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.	

1. See the **`sort (1)`** page in the online *Command Reference* for all available options and an explanation of their capabilities.

5
Overview of the Tutorials

5
Overview of the Tutorials

Introduction

This chapter provides a brief description of the tutorials included in the second part of this book. These tutorials teach you how to do the following:

- edit text (Chapter 6, the "*Line Editor (ed) Tutorial*" and Chapter 7, the "*Screen Editor (vi) Tutoria*l").
- print text (Chapter 8, "*LP Print Service Tutorial*").
- communicate electronically (Chapter 10, the "*Electronic Mail Tutorial*", Chapter 11, the "*Remote Services Tutorial*", and Chapter 12, the "*Communication Tutorial*").

Using the Text Editors

Using the filesystem is a way of life in the UNIX system environment. The section will teach you how to create and modify files using a software tool called a text editor. The section begins by describing a text editor and explaining how it works. Then it introduces two types of text editors supported on the UNIX system:

- the line editor, **`ed`**
- the screen editor, **`vi`** (short for visual editor).

A comparison of the two editors is also included. For detailed information about **`ed`** and **`vi,`** see Chapter 6, and Chapter 7, for the tutorial chapters in this book.

Text Editing Tasks

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)
- 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. To the shell, 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 set of shell commands. 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.

Text Editing Buffers

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

To save your work, you must tell the text editor to write the contents of the buffer into a file; the file is then stored in computer memory. If you don't do this, the contents of the buffer will disappear when you leave the editing program. To prevent this from happening, the text editors send you a reminder to write your file if you attempt to end an editing session without doing so.

NOTE

If you have made a mistake or do not want the edited version, you can choose to leave the editor without writing the file. When you do this, you leave the original file intact, but the edited copy disappears.

Regardless of whether you are creating a new file or updating an existing one, the text in the buffer is organized into lines. A line of text is simply a series of characters that appears horizontally across the screen and is ended when you press the <RETURN> key.

Occasionally, files may contain a line of text that is too long to fit on the terminal screen. Some terminals automatically display the continuation of the line on the next row of the screen; others do not.

Modes of Operation

Text editors are capable of operating in two modes of operation:

- command mode
- 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 text input 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've performed the text editor exercises in Chapter 6, "*Line Editor (ed) Tutorial*", and Chapter 7, the "*Screen Editor (vi) Tutorial.*"

Line Editor

The line editor, accessed by the **ed** command, is used 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 for changing and printing lines, reading and writing files, and entering text.

You can invoke the line editor from a shell program; something you cannot do with the screen editor. (See "*Compilation Systems*" for information on basic shell programming techniques.)

The line editor (**ed**) works well on video display terminals and paper printing terminals. It can also be used across a slow-speed telephone line. (The visual editor, **vi,** can be used only on video display terminals.) Refer to Chapter 6, the "*Line Editor (ed) Tutorial*" for instructions, and Appendix C, "*Quick Reference to ed Commands*" for a summary of line editor commands.

Screen Editor

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

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.

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

The trade-off for the screen editor's speed, visual appeal, efficiency, and power is the heavy demand it places on computer processing time. Every time you make a change, no matter how simple, **`vi`** must update the screen. See the Chapter 7,"*Screen Editor Tutorial*" later in this book, for instructions, and Appendix D, "*Quick Reference to vi Commands*" for a summary of screen editor commands.

Table 5-1 compares the features of the line editor (**`ed`**) and the screen editor (**`vi`**).

Table 5-1. Comparison of Line and Screen Editors (ed and vi)

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

Using the Shell

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

The shell is much like other programs, except that instead of performing one job, as **`cat`** or **`ls`** does, it is central to your interactions with the UNIX system. The primary function of the shell 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.

Various methods of using the shell enhance your ability to use system features. Besides using it to run a single program, you can also use the shell to:

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

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

The shell can be used to control your environment. When you log in to the UNIX system, the shell automatically sets up a computing environment for you. The default environment set up by the shell includes these variables:

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

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

You can use the default environment supplied by your system or you can tailor an environment to meet your needs. If you choose to modify any part of your environment, you can use either of two methods to do so. If you want to change a part of your environment only for the duration of your current computing session, specify your changes in a command line.

If you want to use a different environment (not the default environment) regularly, you can specify your changes in a file that will set up the desired environment for you automatically every time you log in. This file must be called **`.profile`** and must be located in your home directory.

The **`.profile`** typically performs some or all of the following tasks:

- checks for mail
- sets data parameters, terminal settings, and tab stops
- assigns a character or character string as your login prompt
- assigns the erase and kill functions to keys.

You can define as few or as many tasks as you want in your **`.profile.`** You can also change parts of it at any time. (For instructions, see *Compilation Systems.)*

To see whether or not you have a **`.profile:`**

1. If you are not already in your home directory, use the **`cd`** command to get there.
2. Examine your **`.profile`** by issuing this command:

   ```
   cat .profile
   ```
3. 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 *Compilation Systems* for instructions.)

Using the Print Services

After you have created files with a text editor, you may want to print copies on a printer. The UNIX system provides a print service for

- printing files
- controlling the appearance of a finished document
- selecting a printer
- monitoring the print process
- enabling and disabling printers.

The print service supports many different types of printers. When you submit a job to a designated printer, the print service send your print request to a print queue. Your print request is sent to the printer you designate and will be printed using any print options you specify.

Chapter 8, "*LP Print Service Tutoria*l" describes the printing processes and options available to you.

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. Specifically, you can send and receive messages, exchange files, and form networks with other UNIX systems, as long as you're logged in on one UNIX system capable of communicating with others.

This guide introduces you to several communication programs available for exchanging data with other systems. Chapter 10,"*Electronic Mail Tutorial*", Chapter 11,"*Remote Services Tutorial*", and Chapter 12, "*Communication Tutorial*" describe various tools for electronic communication and provide exercises so you can practice using them.

6
Line Editor (ed) Tutorial

6
Line Editor (ed) Tutorial

Introduction

This chapter is a tutorial on the line editor, **ed**. **ed** can be used on any type of terminal. The examples of command lines and system responses described 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.

During editing sessions, **ed** always points 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 a different number or a 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 that file is placed in the buffer and your changes are made to this copy. The changes have no effect on the original file until you instruct **ed**, by using the write command, to move the contents of the buffer into the file.

After you have read this tutorial and tried the examples and exercises, you will know how to:

- enter **ed** (the line editor), create text, write the text to a file, and quit **ed**
- address and display particular lines of the file
- delete text
- substitute new text for old text
- use special characters in search and substitution patterns
- move text around in the file.

Using the Line Editor (ed) 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 Appendix C, "*Quick Reference to ed Commands*,"where the commands are listed by topic.

At the end of some sections, exercises are given so you can experiment with the commands. Answers provided for the exercises in this chapter are not 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.

The notation conventions used in this chapter are the same used throughout this guide. They are described in the "Introduction."

NOTE

Some **`ed`** commands, such as **`G`**, **`P`**, and **`Q`**, and the use of the special pattern matching characters `(`, `)`, `{`, and `}`, are not discussed in this tutorial.

These commands are discussed on the **`ed(1)`** page of the online *Command Reference*. Experiment with these commands to see what tasks they perform.

Getting Started with ed

The best way to learn **`ed`** is to log in to the UNIX system and try the examples as you read this tutorial. Do the exercises; do not be afraid to experiment.

In this section you will learn how to:

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

Entering ed

To enter the line editor, type **`ed`** and a filename:

```
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 filename:

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

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

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

Creating Text Using ed

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

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

```
a <RETURN>
```

You are now in input mode; any characters you type in this mode 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. You can then issue other **ed** 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>
```

ed does not give a response to the period; it just waits for a new command. If **ed** does not respond to a command, you may have forgotten to type a period after entering text and may still be in text input mode.

Type a period and press the <RETURN> key at the beginning of a line to return to command mode. You can now execute editing commands. For example, if you have added some unwanted characters or lines to your text, you can delete them once you have returned to command mode.

Displaying Text with ed

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 the address of the line). 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 (separated by a comma), of the first and last lines of the section you want to see. 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.

```
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:

1. Add another line of text to your **try-me** file.
2. Use the **p** command without quitting text input mode.
3. 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 explains how to delete the unwanted line.

Deleting a Line of Text with ed

To delete text, you must be in the command mode of **ed**. Typing **d** deletes the current line; typing **d** and the line number deletes the specified line. Try this command on the last example to remove the unwanted line containing **p**:

1. Display the current line (**p** command),
2. Delete it (**d** command),
3. 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.
```

Moving Up or Down in a File Using ed

To display the line below the current line, press the <RETURN> key while in command mode. If there is no line below the current line, **ed** responds with a ? and continues to treat the last line of the file as the current line.

To display the line above the current line, press the minus key (-).

The following screen provides examples of how both these commands are used:

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

Notice that by typing - <RETURN> or <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 you want to see the line you have specified. Experiment with these commands:

- create some text
- delete a line
- display your file.

Saving the Buffer Contents in a File

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

By writing to a file, you are putting a copy of the contents of the buffer into the file. The text in the buffer is not disturbed, and you can make further changes to it.

NOTE

You should 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 have to specify a filename; just type **w** and press the <RETURN> key. If you have just created new text, **ed** creates a file for it with the name you specified when you entered the editor. If you 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:

```
w stuff <RETURN>
110
```

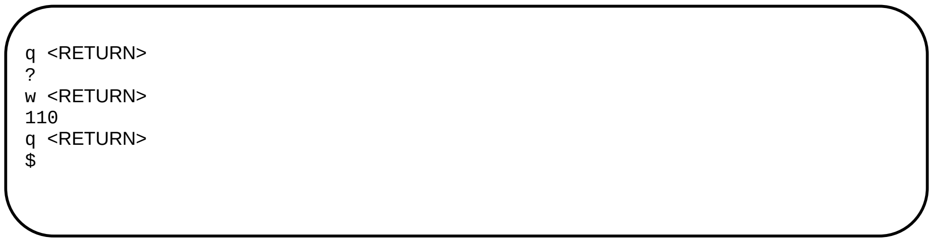
Notice the last line of the screen. This is the number of characters in your text. When the editor reports the number of characters in this way, the write command has succeeded.

Leaving ed

When you have completed editing **try-me**, 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>
110
q <RETURN>
$
```

The system responds with a shell prompt. At this point the editing buffer is discarded. If you have not executed the write command, your text in the buffer has also vanished. If you have not made any changes to the text during your editing session, no harm is done. However, if you have made changes, you can 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>
110
q <RETURN>
$
```

If, instead of writing, you type **q** a second time, **ed** assumes you do not want to write the contents of the buffer 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**.

Table 6-1 summarizes these commands.

Table 6-1. Summary of ed Editor 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 the terminal screen.
d	Delete text.
<RETURN>	Display the next line in the buffer (similar to a carriage return).
-	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.

Getting Started with ed: Exercises

Exercise 1-1:

1. Enter **ed** with a file named **junk**.
2. Create a line of text containing `Hello World.`
3. Write it to the file.
4. Quit **ed**.
5. Use **ed** to create a file called **stuff**.
6. Create a line of text containing two words, `Good-bye world.`
7. Write this text to the file.
8. Quit **ed**.

Exercise 1-2:

1. Enter **ed** again with the file named **junk**. What was the program response? Was the character count for it the same as the character count reported by the **w** command in Exercise 1-1?
2. Display the contents of the file. Is that your file **junk**?
3. 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?

Exercise 1-3:

1. Enter **ed** with the file **junk**.

2. Add a line:

   ```
   Wendy's horse came through the window.
   ```

   Because you did not specify a line address, where do you think the line was added to the buffer?

3. Display the contents of the buffer.
4. Try quitting the buffer without writing to the file.
5. 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.
6. Now type **`q`** to quit. Notice that **`ed`** does not give the `?` warning, even though you typed **`q`** without writing the changes to **`junk`**. The reason for this is that once you write the buffer to a file—any file—**`ed`** no longer considers the buffer modified.

Getting Started with ed: Answers for Exercises

Answers for Exercise 1-1:

```
$ ed junk <RETURN>
?junk
a <RETURN>
Hello world. <RETURN>
. <RETURN>
w <RETURN>
13
q <RETURN>
$
```

```
$ ed stuff <RETURN>
?stuff
a <RETURN>
Goodbye world. <RETURN>
. <RETURN>
w <RETURN>
15
q <RETURN>
$
```

Answers for Exercise 1-2:

```
$ ed junk <RETURN>
13
1,$p <RETURN>
Hello world.
q <RETURN>
$
```

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

Answers for Exercise 1-3:

```
$ ed junk <RETURN>
13
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>
52
q <RETURN>
$
```

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 filename, or another line address.

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

Line Addressing with ed

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; specify 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 Address

`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 lines to different positions. This will change the line address numbers of some lines.

The number of a specific line is always the current position of that line in the editing buffer. For example, if you add five lines of text between lines 5 and 6, line 6 becomes line 11. If you delete line 5, line 6 becomes line 5.

Symbolic 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 by typing a period (**.**) and pressing the <RETURN> key.

Use 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.
```

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

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,$`. Use 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 Current Line through the Last Line

The semi-colon (`;`) represents a set of lines beginning with the current line and ending with the last line of a file. It is equivalent to the symbolic address `.,$`. Use 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.
```

Addresses Relative to the Current Line

You may often want to address lines in relation 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. An address derived in this way is called a relative address. 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 indicates an error. To get a help message that describes the error, see the instructions in *"Other Useful ed Commands and Files"* at the end of this chapter.

Character String Address

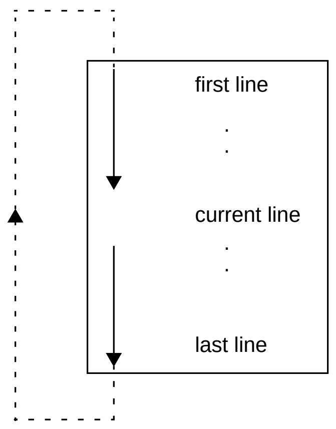
You can search forward or backward in the file for a line containing a particular character string. To do so, specify a string, preceded by a delimiter.

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

/*pattern*

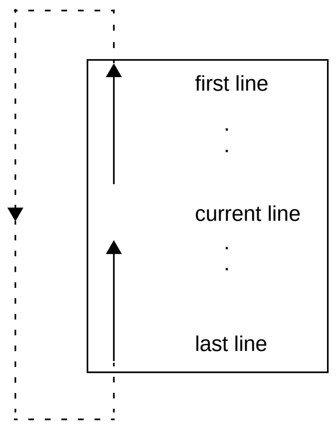
When you specify a pattern preceded by a /, **ed** begins at the current line and searches forward (that is, through subsequent lines in the buffer) for the next line containing the pattern. When the search reaches the last line of the buffer, **ed** wraps around to the beginning of the file and continues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

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



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

```
$ ed try-me <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** prints the first occurrence, it waits for another command. Type `/` and press the `<RETURN>` key; **ed** will continue to search forward through the file for the last pattern specified. Use 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.

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 `2` as *address1* and `7` as *address2* in the following format:

```
2,7p <RETURN>
```

Use this method 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.

You can also use relative line addresses 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
```

Global Search with ed

Two commands 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 *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 (`line`).

Once again, the default command for the lines addressed by **g** or **v** is **p**; you do not have 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 must include beginning and ending delimiters. You can use any of the methods discussed in this section to specify line addresses for **ed** commands. Table 6-2 summarizes the symbols and commands available for addressing lines.

Table 6-2. Summary of ed Line Addressing

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

Table 6-2. Summary of ed Line Addressing (Cont.)

Address	Description
`+` *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*

Line Addressing with ed: Exercises

Exercise 2-1:

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. Display line 3.
3. If you specify a range of lines with the relative address `-2,+3p`, what lines are displayed?
4. What is the current line number? Display the current line.
5. What does the last line say?
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?

7. Search for all lines that contain the pattern `in`. Then search for all lines that do not contain the pattern `in`.

Line Addressing with ed: Answers for Exercise

Answers for Exercise 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><RETURN>
I lost my heart in <RETURN>
San Francisco <RETURN>
I lost $$ in <RETURN>
Las Vegas <RETURN>
. <RETURN>
w <RETURN>
163
```

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

```
-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
```

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

```
$ <RETURN>
Las Vegas
```

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

```
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
```

Displaying and Creating Text with ed

The following **ed** commands display lines of text in the editing buffer:

- **p**
- **n**

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

- **a**
- **i**
- **c**

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 Table 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.

Table 6-3. Sample Addresses for Displaying Text

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** should display the set of lines from line 1 through the first line after the current line that contains the character *x*. It is important to enclose the letter *x* between slashes so **ed** can distinguish between the search pattern address (*x*) and the **ed** command (**p**).

Displaying Text with Line Numbers: 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.

Use **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
```

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 earlier in this tutorial. 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 new file called **new-file** is created. In the first append command line, the default address is the current line. In the second append command line, line 1 is specified as *address1*. The lines are displayed with **n** so 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 <RETURN>
of text in <RETURN>
this file. <RETURN>
. <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 in Table 6-4 allow you to move through and add to the text quickly in this way.

Table 6-4. Append Commands

.a <RETURN>	Append text after the current line.
$a <RETURN>	Append text after the last line of the file.
0a <RETURN>	Append text before the first line of the file (at a symbolic address called line 0),

To use 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 last example refers to the line before line 1 as the line after line 0. To avoid such ambiguous 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>
28
```

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

```
2i <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   This is the beginning.
2   Line 1
3   This is the new 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>
```
or
```
$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 the text input mode. To leave the input mode, type a period alone on a line.

Address1 is the first line and *address2* is the last line of the range of lines to be replaced by new text. To erase one line of text, specify only *address1*. If you do not specify an address, **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 1 through 4 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>
. <RETURN>
This is the new line 5.
```

If you are not sure whether you have left the text input mode, it is a good idea to type another period. If the current line is displayed, you know you are in the command mode of **ed**.

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

Displaying and Creating Text with ed: Exercises

Exercise 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.

Table 6-5. Summary of ed Commands for Displaying and Creating Text

Command	Function
`p`	Display specified lines of text in the editing buffer on the terminal screen.
`n`	Display specified lines of text in the editing buffer with their numerical line addresses on the terminal screen.
`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.

Exercise 3-2

1. Enter **`ed`** with the file **`towns`**. What is the current line?
2. Insert above the third line:

   ```
   Illinois <RETURN>
   ```

3. Insert above the current line:

   ```
   or <RETURN>
   Naperville <RETURN>
   ```

4. Insert before the last line:

   ```
   hotels in <RETURN>
   ```

5. Display the text in the buffer preceded by line numbers.

Exercise 3-3:

1. In the file **`towns`**, display lines 1 through 5 and replace lines 2 through 5 with:

   ```
   London <RETURN>
   ```

2. Display lines 1 through 3.

Exercise 3-4:

1. After you have completed exercise 3-3, what is the current line?
2. Find the line of text containing:

   `Toledo`

3. Replace

   `Toledo`

   with

   `Peoria`

Exercise 3-5:

1. Display the current line.
2. With one command line search for

   `New York`

   and replace it with

   `Iron City`

Displaying and Creating Text with ed: Answers for Exercises

Answers for Exercise 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 where text can be inserted.

Answers for Exercise 3-2:

```
$ ed towns <RETURN>
163
.n <RETURN>
10   Las Vegas
3i <RETURN>
Illinois <RETURN>
. <RETURN>
.i <RETURN>
or <RETURN>
Naperville <RETURN>
. <RETURN>
$i <RETURN>
hotels in <RETURN>
. <RETURN>
1,$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
```

Answers for Exercise 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 nowhere at all in
```

Answers for Exercise 3-4:

```
. <RETURN>
Like being nowhere at all in
/Tol <RETURN>
Toledo
c <RETURN>
Peoria <RETURN>
. <RETURN>
. <RETURN>
Peoria
```

Answers for Exercise 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.

Deleting Text and Undoing Changes with ed

This section discusses commands for deleting text and undoing changes in **ed**. You may use them only when you are working in command mode. The command **d** deletes lines and **u** undoes the changes made by the last command.

Deleting Lines in Command Mode: The d Command

You have already deleted lines of text with the delete command (**d**) earlier in this tutorial.

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 1 through 5 and then deletes lines 2 through 4:

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

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

$**d** <RETURN>

How can you delete the current line? Because one of the most common errors in **ed** is forgetting to type a period to leave the text input mode, 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 Changes in Command Mode: 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** responds with a ? because no more lines are 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
```

NOTE

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

Table 6-6 summarizes the **ed** commands and keys used to delete text in **ed**.

Table 6-6. Summary of ed Commands for Deleting Text

Command	Function
d	Delete one or more lines of text.
u	Undo the previous command.

Substituting Text with ed

You can modify your text with the 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. By default, if an addressed line contains more than one occurrence of *old_text*, only the first occurrence on that line is replaced.
/new_text	The argument specifying the text to replace *old_text*. It is delimited by slashes or by 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 every occurrence of *old_text* on each specified line.
l	Display the last line of substituted text, including nonprinting characters. (See the last section of this chapter, "*Other Useful ed Commands and Files.")*"
n	Display the last line of the substituted text preceded by its numerical line address.
p	Display the last line of substituted text.

Substituting Text on the Current Line

The simplest example of the substitute command is making a change to the current line. You do not have 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>
. <RETURN>
.p <RETURN>
In the beginning, I made an airor.
s/ai/er/ <RETURN>
```

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 a 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 Text 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 Text 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, it makes no changes 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.

Substituting Text Globally

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 each specified line. 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 that 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>`

NOTE

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.

Substituting Text with ed: Exercises

Exercise 4-1:

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
   ```

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`**.

3. Try searching backward in the file for the word

   ```
   lost
   and substitute
   found
   using ? as a delimiter. Did it work?
   ```

4. Search forward in the file for

   ```
   no
   and substitute
   NO
   ```

   for it. What happens if you try to use `?` as a delimiter?

5. 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?

Substituting Text with ed: Answers for Exercises

Answers for Exercise 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.

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

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

```
/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**.

Pattern-Matching Characters in ed

If you try to substitute the `$` sign in the line

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

Instead of replacing the `$`, the new text is placed at the end of the line. The `$` is a special character in **ed** that represents the end of the line.

ed assigns special meanings to several characters as a way of providing a shorthand for search and substitution patterns; these characters are not interpreted literally by the shell. If you specify a special character in a search or substitution pattern, without understanding its special meaning, you will not get the expected results.

Table 6-7 summarizes the special characters for search or substitution patterns.

In the following example, **ed** searches for any three-character sequence ending in the pattern `at.`

Table 6-7. ed Special Characters and Their Meanings

Character	Meaning
`.`	Match any one character.
`*`	Match zero or more occurrences of the preceding character or expression in brackets.
`.*`	Match zero or more occurrences of any character.
`^`	Match the beginning of the line.
`$`	Match the end of the line.
`\`	Take away the meaning of the special character that follows.
`%`	Substitute the last replacement pattern.
`&`	Substitute the text matched by the substitution pattern in the replacement string.
`[...]`	Match the first occurrence of any character in the brackets.
`[^...]`	Match the first occurrence of any character that is not in the brackets.

```
1,$p <RETURN>
rat
cat
turtle
cow
goat
g/.at <RETURN>
rat
cat
goat
```

The word `goat` is included because the string `oat` matches the pattern `.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
```

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 particularly helpful when you want to insert the same pattern at the beginning 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 change text without repeating the text in the replacement string, the & (ampersand) provides a useful shortcut. The & is replaced with the text matching the 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 by using 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,$p <RETURN>
Monday33,000
Tuesday75,000
Wednesday88,000
Thursday62,000
1,$s/[6789]/4 <RETURN>
Monday33,000
Tuesday45,000
Wednesday48,000
Thursday42,000
```

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; that is, 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 in substitution patterns, use a distinctive 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` on every line. Try it.

Experiment with these special characters. Find out what happens when you use them in different combinations.

Table 6-7 summarizes the special characters for search or substitution patterns.

Pattern-Matching Characters in ed: Exercises

Exercise 5-1:

1. Create a file containing the following lines of text.

```
A   Computer Science
D   Jogging
C   Tennis
```

2. What happens if you enter this command:

```
1,$s/[^AB]/A/ <RETURN>
```

3. 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!

4. Insert the following line above line 2:

```
These are not really my grades.
```

5. Using brackets and the `^` character, create a search pattern 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.

Exercise 5-2:

1. Create a file containing the following lines:

```
I love money
I need money
The IRS wants my money
```

2. Now use one command to change them to:

```
It's my money
It's my money
The IRS wants my money
```

3. Using two command lines, do the following:

   a. change the word on the first line from `money` to `gold`.

   b. Change the last two lines from `money` to `gold` without using the words `money` or `gold` themselves.

4. How can you change the line

```
1020231020
```

to

```
10202031020
```

without repeating the old digits in the replacement pattern?

Exercise 5-3:

1. Create a line of text containing the following characters.

   ```
   * . \ & % ^ *
   ```

2. Substitute a different letter for each character. Do you have to use a backslash for every substitution?

Pattern-Matching Characters in ed: Answers for Exercises

Answers for Exercise 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>
AA  Computer Science
A  Jogging
A  Tennis
u <RETURN>
```

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

```
2i <RETURN>
These are not really my grades. <RETURN>
. <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
```

Answers for Exercise 5-2:

```
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
```

```
1s/money/gold <RETURN>
It's my gold
2,$s//% <RETURN>
The IRS wants my gold
```

```
s/10202/&0 <RETURN>
10202031020
```

Answers for Exercise 5-3:

```
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 special characters only in the replacement text.

```
s/ \^/g <RETURN>
a c d e f g b
```

Moving and Copying Text with ed

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.

Moving Lines of Text: The m Command

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	The line after which the text will appear after it has been moved.

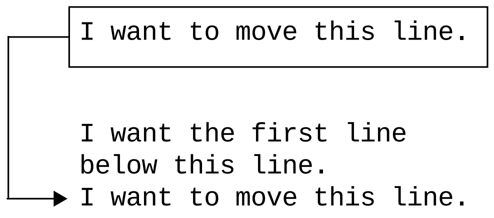
Try the following example to see how the command works. Create a file that contains these three lines of text:

```
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.

Copying Lines of Text: The t Command

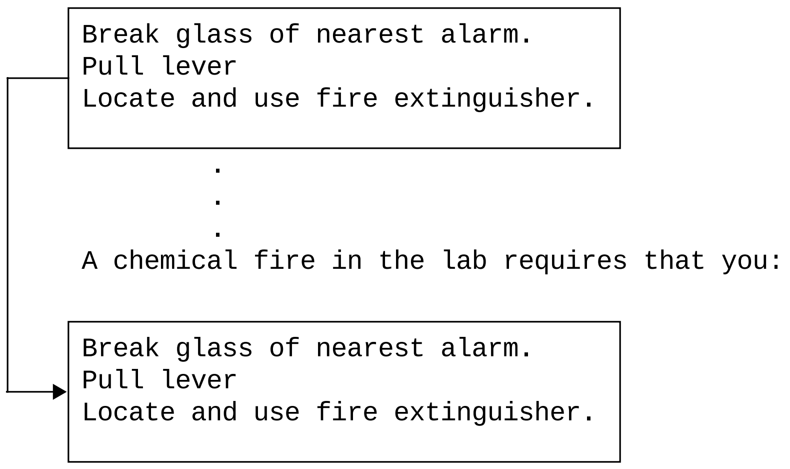
The copy command **`t`** (transfer) acts like the **`m`** command except that the block of text is not deleted at the original address of the line. A copy of that block of text is placed after a specified line of text.

The general format of the **`t`** command resembles that of 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	The line after which the text will appear after it has been moved.

The next example shows how to copy three lines of text below the last line.



```
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

The **j** command joins contiguous lines. The general format is:

[*address1*,*address2*]j <RETURN>

The components of this command line include:

address1,address2
The range of lines to be joined. If no address is given, the current line is joined with the following line. If exactly one address is given, the command does nothing.

`j` The join command.

The next example shows how to join two lines.

```
1,2p <RETURN>
Now is the time to join
the team.
1,2j <RETURN>
1p <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`). You must place a space between them by using the **`s`** command.

Writing Lines of Text to a File: The w Command

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, several lines of a letter are saved in a file called **`memo`**.

```
1,$n <RETURN>
1         March 20, 1991
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.
6   Please plan to attend.
7   Other divisions and locations
8   will also be represented.
9   We will discuss plans
10  for marketing several
11  new products during the
12  coming fiscal year,
13  as well as long range
14  research activities that
15  should yield profitable products
16  during the next decade.
3,5w  memo <RETURN>
91
```

The **w** command places a copy of lines 3 through 5 into a new file called **memo**. **ed** responds with the number of characters in the new file.

The **w** command overwrites preexisting files; it erases the current file and puts the new 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 ed Commands and Files"* later in this chapter, you will learn how to execute shell commands from **ed**. Then you can list the filenames in the directory to make sure 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 5) will be erased and the file will contain only the new lines (13 through 16). However, you can use the **W** command to solve this problem. It will write the current **ed** buffer to the end of the file.

Reading in Files: The r Command

The **r** (read) 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 line after which the text is added. 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,$n <RETURN>
1            March 20, 1991
2   Dear Michael,
3   Are you free later today?
4   Hope to see you there.
3r memo <RETURN>
91
3,$n <RETURN>
3   Are you free later today?
4   There will be 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 they are correct.

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

Table 6-8. Summary of 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.
W	Append text to an existing file.
r	Read in text from another file.

Moving and Copying Text with ed: Exercises

Exercise 6-1:

1. There are two ways to copy lines of text in the buffer:

   - by issuing the copy command
   - by using the write and read commands to write text to a file first 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?

2. What commands can you use to copy lines 10 through 17 of file **exer** into a file called **exer6** at line 7?

3. Lines 33 through 46 give an example that you want placed after line 3, and not after line 32. What command performs this task?
4. 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?

Moving and Copying Text with ed: Answers for Exercises

Answers for Exercise 6-1:

Any time you have lines of text that must appear in several places in the same file, you can save time by writing those lines to a separate file and reading in that file at the appropriate points in the file you're editing.

Similarly, if you want to copy a set of lines from the current file to another one, write the desired lines to a separate 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
```

In this example, the temporary file happens to be called **`temp.`**

```
33,46m3 <RETURN>
```

```
13,14j <RETURN>
```

Other Useful ed Commands and Files

Four other commands and a special file may be useful to you during editing sessions.

`h,H`	Access the help commands, which provide error messages.
`l`	Display characters that are not normally displayed.
`f`	Display the current filename.
`!`	Temporarily escape **`ed`** to execute a shell command.
`ed.hup`	A special file in which the contents of the buffer are saved if **`ed`** is interrupted.

Getting Help: The h and H Commands

You may notice, when 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. Two help commands allow you to request brief explanations when you get a diagnostic message.

`h`	Display a short error message that explains the reason for the most recent `?`.
`H`	Place **`ed`** in help mode so a short error message will be displayed every time the `?` appears. (To cancel this command, type **`H`** again.)

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>
UX: ed: WARNING: expecting `w'
```

The `?` is also displayed when you specify a new filename on the **`ed`** command line. Give **`ed`** a new filename. When the `?` appears, type **`h`** to find out what the error message means.

```
ed newfile <RETURN>
?newfile
h <RETURN>
UX: ed: ERROR:Cannot open input file
```

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

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

```
$ ed newfile <RETURN>
H <RETURN>
UX:ed:ERROR:Cannot open input file
/hello <RETURN>
?
UX:ed:ERROR:Search string not found
1,22p <RETURN>
?
UX:ed:ERROR:Line out of range
a <RETURN>
I am appending this line to the buffer. <RETURN>
. <RETURN>
s/$ tea party <RETURN>
?
UX:ed:ERROR:Illegal or missing delimiter
,$s/$/ tea party <RETURN>
?
UX:ed:ERROR:Unknown command
H <RETURN>
q <RETURN>
?
h <RETURN>
UX:ed:WARNING:Expecting `w'
```

Some of the most common error messages you may encounter during editing sessions are:

`search string not found`	**ed** cannot find an occurrence of the search pattern `hello`.
`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 need to see error messages only occasionally. Then you can use the **h** command.

Displaying Nonprinting Characters: The l Command

Control characters usually do not appear on the terminal screen. For example, <`CTRL`><`g`> (control-g) rings the terminal bell but does not appear on the screen.

If you enter a <`TAB`> character, the terminal normally displays up to eight spaces covering the space up to the next tab setting. (Your tab setting may be fewer or more than eight spaces. For details, see the discussion of **`stty`** in *Compilation Systems Manual.)*

If you want to see how many tabs or control characters 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 is displayed. If only *address1* is given, only that line is displayed.
`l`	The command that displays the nonprinting characters along with the text.

The **`l`** command displays tabs with a `>` (greater than) character. To enter a control character, hold down the <`CTRL`> (control) key and press the appropriate alphabetic key. The key that sounds the bell is <`CTRL`><`g`> (control-g). It is displayed as `\007` which is the octal representation (the computer code) for control-g.

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

```
a <RETURN>
Add a control-g to this line. <RETURN>
Add a tab to this line. <RETURN>
. <RETURN>
1,2l <RETURN>
Add a \007 (control-g) to this line.
Add a > (tab) to this line.
```

Did the bell sound when you typed <`CTRL`><`g`> ?

Displaying the Current Filename: The f command

In a long editing session, you may forget the filename. The **`f`** command will remind you which file is currently in the buffer. Or, you may want to preserve the file you're now editing as it existed at the beginning of your editing session, and write the contents of the buffer to a new file. To avoid accidentally overwriting the original file with the customary **`w`** and **`q`** command sequence, you can tell the editor to associate the contents of the buffer with a new filename while you are in the middle of the editing session. Do this with the **f** command and a new filename.

The format for displaying the current filename 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 filename use this general format:

f *newfile* <RETURN>

If you do not specify a filename with the write command, **ed** remembers the filename 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 filename with the write command, or change the current filename using the **f** command, followed by the new filename. Because you can use **f** at any point in an editing session, you can change the filename 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.

Escaping to the Shell: The ! Command

How can you make sure you are not overwriting an existing file when you write the contents of the editor to a new filename? You must return to the shell to list your files. The **!** allows you to return temporarily 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 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 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>
Mon  Apr 1  14:24:22  EST  1991
!
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 *Compilation Systems Manual.*

Recovering from Hangups: The ed.hup File

What happens if you are creating text in **ed** and the line connecting your terminal to the computer is disconnected, or your terminal is unplugged? When a hangup 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 filename, such as the name the file had while you were editing it (before the hangup). 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.

Table 6-9 summarizes the functions of the commands and the special file introduced in this section.

Table 6-9. Summary of Other Useful ed Commands and Files

Command	Function
`h`	Display a short error message for the preceding diagnostic `?`.
`H`	Turn on help mode so an error message is given for each diagnostic `?` When **`H`** is entered a second time: Turn off help mode.
`l`	Display nonprinting characters in the text.
`f`	Display the current filename.
`f` *newfile*	Change the current filename associated with the editing buffer to *newfile*.
`!`*cmd*	Temporarily escape to the shell to execute a shell command **`cmd`**.
`ed.hup`	A special file in which buffer contents are saved when **`ed`** is interrupted.

Other Useful ed Commands and Files: Exercises

Exercise 7-1:

1. Create a new file called **`newfile1`**.
2. Access **`ed`** and change the name of the file to **`current1`**.
3. Create some text and write and quit **`ed`**.
4. Run the **`ls`** command to verify that a file called **`newfile1`** does not exist in your directory.

Exercise 7-2:

1. Create a file called **`file1`**.
2. Append some lines of text to the file.

3. Leave append mode, but do not write the file.
4. Turn off your terminal. Then turn on your terminal and log in again.
5. Issue the **`ls`** command in the shell. Is there a new file called **`ed.hup`**?
6. Edit **`ed.hup`** using **`ed`**. How can you change the current filename to **`file1`**?
7. Display the contents of the file. Are the lines the same lines you created before you turned off your terminal?

Exercise 7-3:

While you are in **`ed`**, temporarily escape to the shell and send a mail message to yourself.

Other Useful ed Commands and Files: Answers for Exercises

Answers for Exercise 7-1:

```
$ ed newfile1 ?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
```

Answers for Exercise 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?
```

Answers for Exercise 7-3:

```
$ ed file1 <RETURN>
58
! mail mylogin <RETURN>
You will get mail when <RETURN>
you are done editing! <RETURN>
. <RETURN>
!
```

7
Screen Editor (vi) Tutorial

7
Screen Editor (vi) Tutorial

Introduction

This chapter is a tutorial on the screen editor, **vi** (visual). The **vi** editor is a powerful and sophisticated tool for creating and editing files. It is designed for use with a video display terminal which is used as a window through which you can view the text of a file. A few simple commands allow you to make changes to the text that are quickly reflected on the screen.

The **vi** editor displays from one to many lines of text. It allows you to move the cursor to any point on the screen or in the file (by specifying places such as the beginning or end of a word, line, sentence, paragraph, or file) and create, change, or delete text from that point. You can also use **ex** line editor commands, such as the powerful global commands that allow you to change multiple occurrences of the same character string by issuing one command.

To move through the file, you can scroll the text forward or backward, revealing the lines below or above the current window, as shown in Figure 7-1.

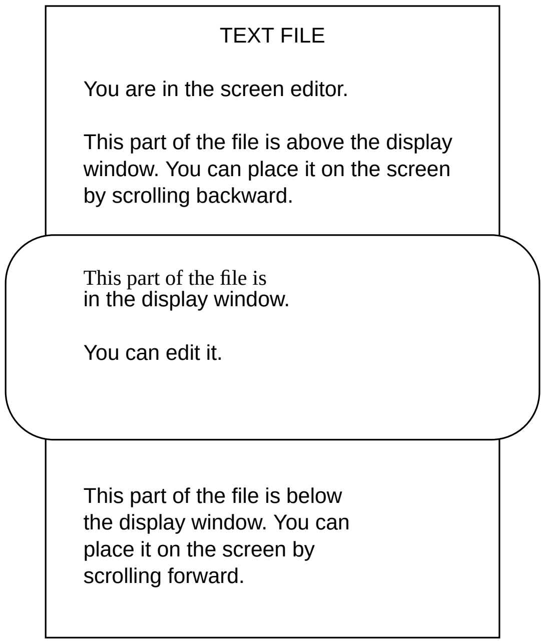

TEXT FILE
You are in the screen editor.
This part of the file is above the display window. You can place it on the screen by scrolling backward.
This part of the file is in the display window.
You can edit it.
This part of the file is below the display window. You can place it on the screen by scrolling forward.

Figure 7-1. Displaying a File with a vi Window

NOTE

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

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:

- change your shell environment to set the configuration of your terminal
- enable automatic carriage return
- 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
- use **ex** line editing commands from within **vi**
- temporarily escape to the shell to execute shell commands
- recover a file lost by an interruption to an editing session
- edit several files in the same session.

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

The commands discussed in each section are reviewed at the end of the section. A summary of **vi** commands is found in Appendix D "*Quick Reference to vi Commands*", where they are listed by topic.

At the end of some sections, exercises are given so you can experiment. The best way to learn **vi** is by following the examples and doing the exercises as you read the tutorial. The answers given for the exercises are suggested ways of doing the exercises. Keep in mind that there is often more than one way to perform a task in **vi**. If your method works, it is correct.

Log in on the UNIX system when you are ready to read this chapter.

Getting Started with vi

Before you enter **vi**, you must set your terminal configuration. Because the software for the **vi** editor is executed differently on different terminals, you must tell the UNIX system what type of terminal you are using.

Each type of terminal has several code names that are recognized by the UNIX system. In the appendix, “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 supported on your UNIX system. It is a good idea to check with your local system administrator for the most up-to-date list of available terminal types.

There are two different options for specifying the terminal type:

- if you use different terminals to get access to the UNIX system, you will need to set the terminal (TERM) environment variable every time you log in
- if you always use the same terminal type, you can set the TERM variable in a file in your login directory once, not every time you log on.

Setting the Terminal Type for a Single Login Session

To set your terminal configuration, immediately after you log on 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; if you don't, terminal functions such as tab settings may not work properly.

For example, if your terminal is an AT&T 630 this is how your commands will appear on the screen.

```
$ TERM=630 <RETURN>
$ export TERM <RETURN>
$ tput init <RETURN>
```

Do not experiment by entering names for terminal types other than your terminal. This may cause **vi** to display the screen incorrectly.

Setting the Terminal Type for All Sessions

If you are going to use the same terminal type 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**.

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 three lines shown in the last screen (the TERM assignment, **export** command, and

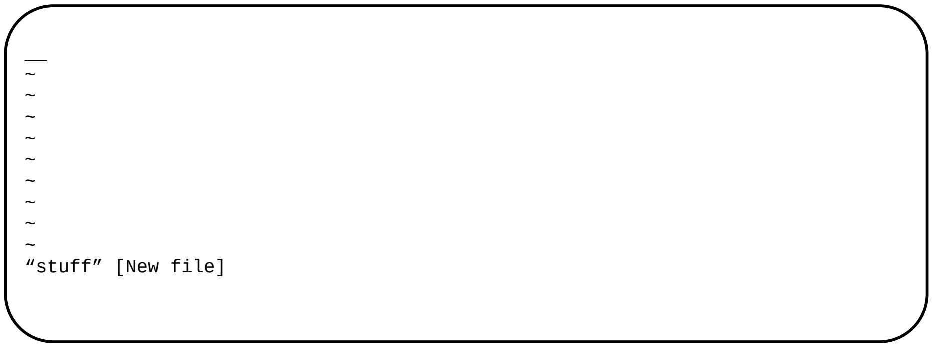
tput command) to your **.profile**. For detailed instructions, see Chapter 9, the "*Shell Tutorial*" chapter.

Entering vi

First, enter the editor; type **vi** and the name of the file you want to create or edit.

```
vi filename <RETURN>
```

For example, suppose you want to create a file called **stuff**. When you type the **vi** command with the filename **stuff**, **vi** clears the screen and displays a window in which you can enter and edit text.



```
__
~
~
~
~
~
~
~
~
~
"stuff" [New file]
```

The __ (underscore) on the top line shows the cursor. 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 you forgot to set your terminal configuration or you set it to the wrong type of terminal before entering **vi**, you will see an error message instead.

```
$ vi stuff <RETURN>
terminal_name: Unknown terminal type
I don't know what kind of terminal you are on - All I have is "unknown"
[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.

vi Operating Modes

The **vi** editor operates in two modes:

- insert (text input) mode

- command mode.

In the insert mode you can add and modify text; in the command mode you can:

- edit and change existing text
- delete, move, and copy text
- move around in the file
- perform other tasks.

Insert Mode: Creating and Adding Text with vi

If you have successfully entered **`vi`**, you are in the command mode and **`vi`** is waiting for your commands. To create text

1. Type an **`a`** to enter the insert mode of **`vi`**. Do not press the **<**RETURN**>** key. The **`a`** is not printed on the screen.
2. You can now add text to the file. Type some text.

Leaving Insert Mode

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

```
Create some text <RETURN>
in the screen editor <RETURN>
and return to <RETURN>
command mode. <ESC>
~
~
```

If you press the <ESC> key and a bell sounds, you are already in the command mode. Pressing the <ESC> key while you are in the command mode does not affect the text in the file, even if you do it several times.

NOTE

On some types of terminals, **`vi`** flashes the screen instead of sounding the bell.

Command Mode: Editing Text in vi

To edit an existing file you must be able to add, change, and delete text. However, before you can perform those tasks you must be able to move to the part of the file you want to edit. **`vi`** offers an array of commands for moving from page to page, between lines, and between specified points inside a line. These commands, along with commands for deleting and adding text, are introduced in this section.

Command Mode: Moving the Cursor

To edit your text, you need to move the cursor to the point on the screen where you will begin the correction. This is easily done with four keys that are grouped together on the keyboard: **`h`**, **`j`**, **`k`**, and **`l`**. See Figure 7-2.

`h`	Move the cursor one character to the left.
`j`	Move the cursor down one line.
`k`	Move the cursor up one line.
`l`	Move 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.

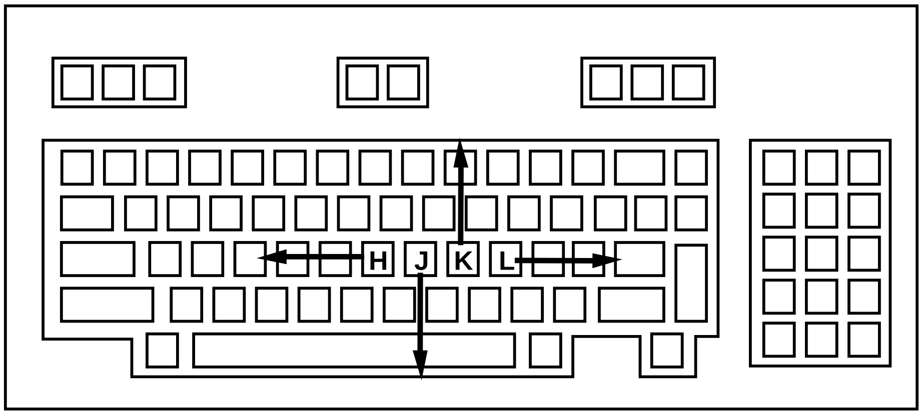


Figure 7-2. Keys That Move the Cursor

NOTE

Some terminals have special cursor control keys that are marked with arrows. Use them in the same way you use the **`h`**, **`j`**, **`k`**, and **`l`** commands.

Watch the cursor on the screen while you press the **h**, **j**, **k**, and **l** keys. 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** sounds 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 fewer 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 fewer 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 insert mode and are adding text to your file. If the motion command letters appear on the screen, you are still in the insert mode. Press the <ESC> key to return to the command mode and try the commands again.

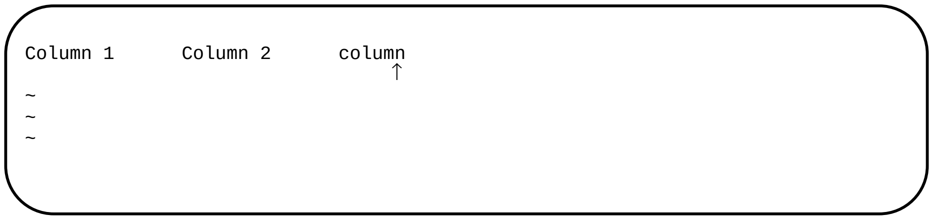
In addition to the motion command keys **h** and **l**, the <SPACEBAR> and <BACKSPACE> keys can be used to move the cursor right or left to a character on the current line. Refer to the following table.

Table 7-1. Cursor Movement Keys

<SPACEBAR>	Move the cursor one character to the right
n <SPACEBAR>	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.

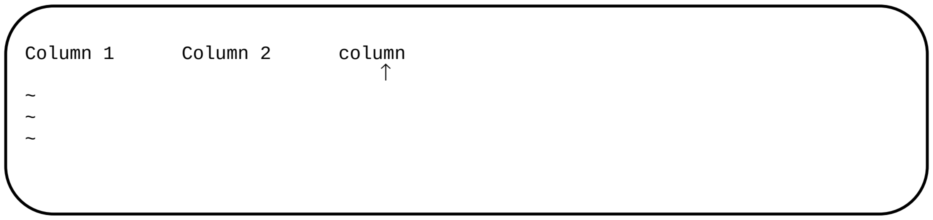
To move the cursor quickly to the right or left, type a number before the command. For example, suppose you want to create four columns on your screen and after you finish typing the headings for the first three columns, you notice a typing mistake, as shown in the screen below.



```
Column 1      Column 2      column
                                 ↑
~
~
~
```

You want to correct your mistake now, before you continue typing. Exit the insert mode and return to the command mode by pressing the <`ESC`> key; the cursor will move to the `n`.

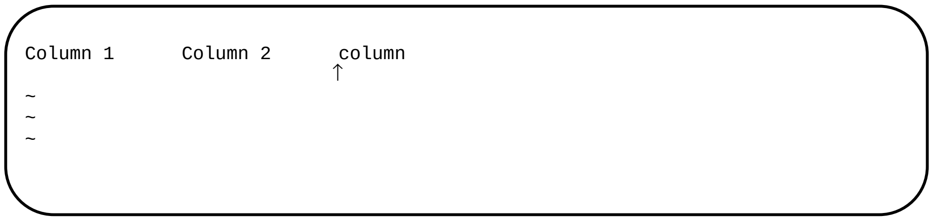
```
<ESC>
```



```
Column 1      Column 2      column
                                ↑
~
~
~
```

Then use the **`h`** command to move back five spaces.

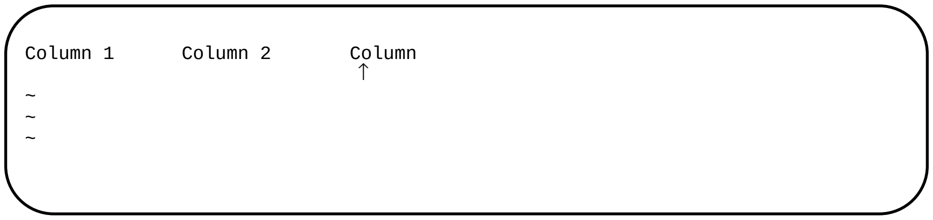
```
5h
```



```
Column 1      Column 2      column
                            ↑
~
~
~
```

Erase the `c` by typing **`x`**. Then change to insert mode (with **`i`**), enter a `C`, and press the <`ESC`> key.

```
xiC <ESC>
```



```
Column 1      Column 2       Column
                              ↑
~
~
~
```

Use the **`l`** motion command to return to your earlier position.

```
5l
```

```
Column 1      Column 2       Column
                                  ↑
~
~
~
```

The **x** and **i** commands are discussed in detail below.

Command Mode: Deleting Text

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

x	Delete one character.
*n***x**	Delete *n* characters, where *n* is the number of characters you want to delete.

```
Hello wurld!
       ↑
~
~
~
```

x

```
Hello wrld!
      ↑
~
~
~
```

Now try preceding **x** with the number of characters you want to delete. For example, delete the second occurrence of the word `deep` from the text shown in the following screen. Put the cursor on the first letter of the string you want to delete, and delete five characters (for the four letters of `deep` plus an extra space).

```
Tomorrow the Loch Ness monster
shall slither forth from
the deep dark deep depths of the lake.
             ↑
~
~
~
```

```
5x
```

```
Tomorrow the Loch Ness monster
shall slither forth from
the deep dark depths of the lake.
             ↑
~
~
~
```

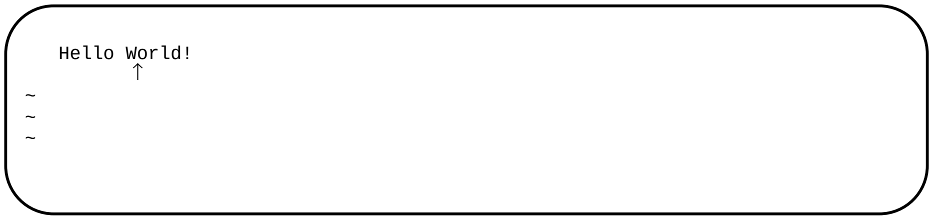
Notice that **`vi`** adjusts the text so that no gap appears in place of the deleted string. If, as in this case, the string you want to delete happens to be a word, you can also use the **`vi`** command for deleting a word. This command is described later in the section *"Positioning the Cursor on a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

Command Mode: Adding Text

There are two basic commands for adding text: the insert (**`i`**) and append (**`a`**) commands. To add text with the insert command at a point in your file that is visible on the screen, move the cursor to that point by using the **`h`**, **`j`**, **`k`**, and **`l`** commands. 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 the `<ESC>` key. If necessary, the original characters will even wrap around onto the next line.

```
   Hello Wrld!
          ↑
~
~
~
```

```
io
```



```
   Hello World!
         ↑
~
~
~
```

To leave insert mode, press

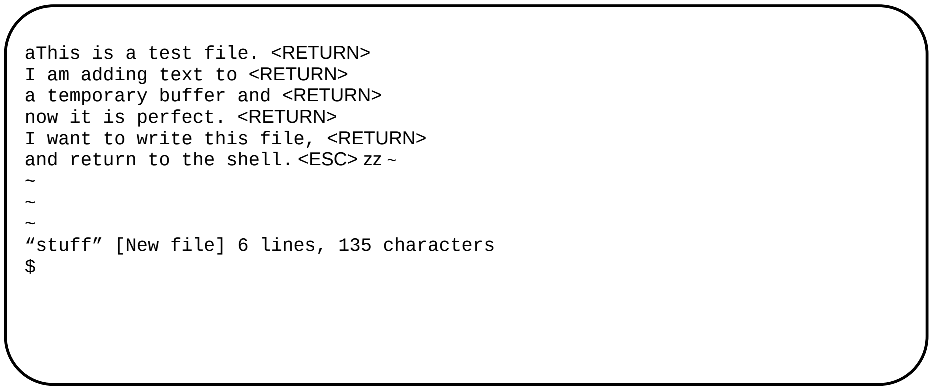
<ESC>

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.

Leaving 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 filename 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 filename and the number of lines and characters in the file. Then the shell gives you a prompt.



```
aThis 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. <ESC> zz ~
~
~
~
“stuff” [New file] 6 lines, 135 characters
$
```

You can also use the **`:w`** and **`:q`** line editor (**`ex`**) commands for writing and quitting a file. Line editor (**`ex`**) 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`**.

```
aThis 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. <ESC>
~
~
~
~
:wq <RETURN>
“stuff” [New file] 6 lines, 135 characters
$
```

Table 7-2 summarizes the basic commands you need to enter and use **vi**.

Table 7-2. Summary of vi Editor Commands

Command	Function
TERM=*terminal_name* **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>	Enter a newline (carriage return).
<ESC>	Leave insert 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 changes to a file and quit **vi**.

Getting Started with vi: Exercises

Exercise 1-1:

As you give commands in the following exercises, watch the screen to see how it changes or how the cursor moves.

1. If you have not logged in yet, do so now. Then set your terminal configuration.
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
   ```

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.
4. Delete the seventh and eighth characters from the right.
5. Move the cursor to the last character on the last line of the text.
6. Append the following new line of text:

   ```
   and byte by byte
   ```

7. Write the buffer to a file and quit **`vi`**.

Exercise 1-2:

1. Reenter **`vi`** and append two more lines of text to the file **`exer1`**.
2. What does the notice at the bottom of the screen say once you have reentered **`vi`** to edit **`exer1`**?

Getting Started with vi: Answers for Exercises

Answers for Exercise 1-1:

Ask your system administrator for your terminal's system name. Type:

```
TERM=terminal_name <RETURN>
export TERM <RETURN>
tput init <RETURN>
```

Enter the **`vi`** command for a file called **`exer1`**:

```
vi exer1 <RETURN>
```

Then use the **`a`** (append) command 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 <RETURN>
~
~
```

Use the **k** and **h** commands.

Use the **x** command.

Use the **j** and **l** commands.

Use the **a** (append) command to enter the following text:

```
<RETURN>
and byte by byte <ESC>
```

Type:

```
ZZ
```

Answers for Exercise 1-2:

Type:

```
vi exer1 <RETURN>
```

Notice the system response:

```
“exer1” 6 lines, 100 characters
```

Positioning the Cursor and Scrolling with vi

Until now you have been moving the cursor with the **h**, **j**, **k**, and **l** commands, the <BACKSPACE> key, and the <SPACEBAR>. There are several other commands that can help you move the cursor quickly around the screen. This section explains how to position the cursor by

- characters on a line
- lines
- text objects
  - words
  - sentences
  - paragraphs

- window.

There are also commands that position the cursor within parts of the **`vi`** editing buffer that are not visible on the screen. These commands will be discussed in the next section.

To follow this section of the tutorial, you should enter **`vi`** with a file that contains at least forty lines. If you do not have a file of that length, create one now. Remember, to execute the commands described here, you must be in command mode of **`vi`**. Press the <`ESC`> key to make sure you are in command mode rather than insert mode.

Positioning the Cursor on a Character

Three ways to position the cursor on a character in a line are by

- moving the cursor right or left to a character
- specifying the character at either end of the line
- searching for a character on a line.

The first method was discussed earlier in this chapter. The following sections describe the other two methods.

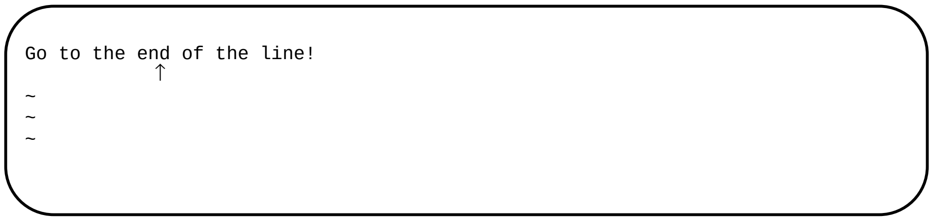
Moving to the Beginning or End of a Line

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

`$`	Put the cursor on the last character of a line.
`0` (zero)	Put the cursor on the first character of a line.
`^` (circumflex)	Put 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.

- Example 1:



```
Go to the end of the line!
            ↑
~
~
~
```

`$`

```
Go to the end of the line!
                         ↑
~
~
~
```

- Example 2:

```
Go to the beginning of the line!
                               ↑
~
~
~
```

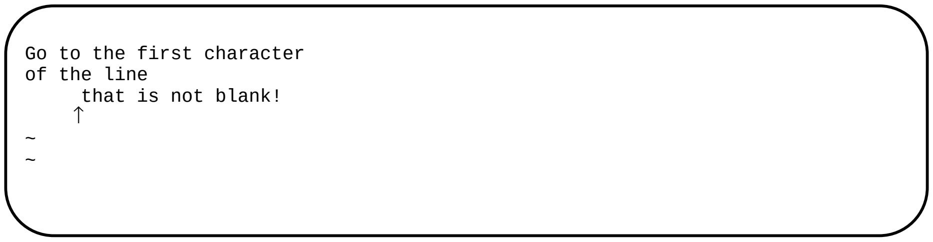
`0`

```
Go to the beginning of the line!
↑
~
~
~
```

- Example 3:

```
Go to the first character
of the line
     that is not blank!
                    ↑
~
~
~
```

`^`



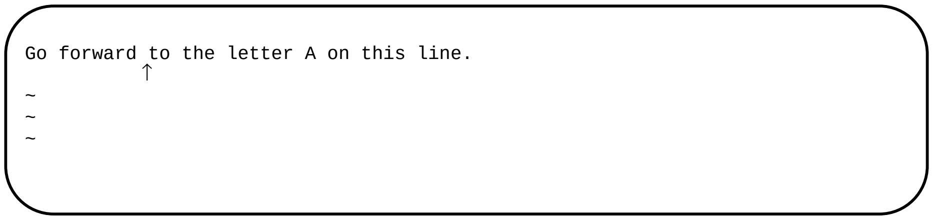
```
Go to the first character
of the line
     that is not blank!
    ↑
~
~
```

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. You can use six commands 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. (Works only with f and t.).
`,`	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. (Works only with f and t.).

For example, in the following screen **`vi`** searches to the right for the first occurrence of the letter A on the current line.



```
Go forward to the letter A on this line.
          ↑
~
~
~
```

```
fA
```

```
Go forward to the letter A on this line.
                         ↑
~
~
~
```

Try the search commands on one of your files.

Positioning the Cursor on a Line

Besides the `j` and `k` keys, which you've already used, the `+`, `-`, and `<RETURN>` keys can be used to move the cursor to other lines.

Moving Up One or More Lines

The **-** command moves the cursor up a line, positioning it at the first nonblank character, if there is one, 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. This is a rapid way to move quickly up a file.

Now try to move up 100 lines. Type:

```
100-
```

What happened to the window? If there are fewer 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.

Moving Down One or More Lines

The plus sign (**+**) or `<RETURN>` key moves the cursor down a line to the first non-blank character. Specify the number of lines you want to move before the **+** command. For example, to move the cursor down nine lines, type:

```
9+
```

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

Now try to do the same thing by pressing the `<RETURN>` key. Are the results the same as they were when you pressed the **+** key?

Positioning the Cursor on a Word

The **`vi`** editor considers a word to be a string of characters that may include letters, numbers, or underscores. There are six commands for positioning words: **`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 white space 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 white space, which can be created by entering spaces, tabs, or newlines.

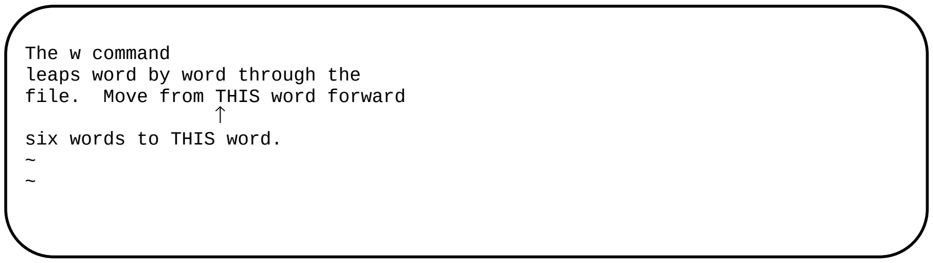
The following is a summary of the word positioning commands.

`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`** command.

*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.

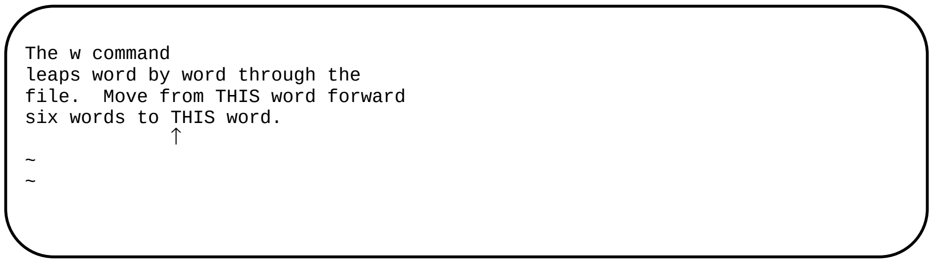
Usage of word positioning commands is shown in the following examples:

- Example 1:



```
The w command
leaps word by word through the
file.  Move from THIS word forward
                 ↑
six words to THIS word.
~
~
```

`6w`



```
The w command
leaps word by word through the
file.  Move from THIS word forward
six words to THIS word.
             ↑
~
~
```

`W` Ignore all punctuation and move the cursor forward to the word after the next blank.

`e` Move the cursor forward in the line to the last character in the next word.

- Example 2:

```
Go forward one word to the end of
the next word in this line
  ↑
~
~
```

e

```
Go forward one word to the end of
the next word in this line
       ↑
~
~
```

- Example 3:

```
Go to the end of the third word after the current word.
       ↑
~
~
```

3e

```
Go to the end of the third word after the current word.
               ↑
~
~
```

E Ignore all punctuation except white space, delimiting words only by white space.

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.

- Example 4:

```
Leap backward word by word through
the file. Go back four words from here.
                             ↑
~
~
```

4B or 4b

```
Leap backward word by word through
the file. Go back four words from here.
          ↑
~
~
~
```

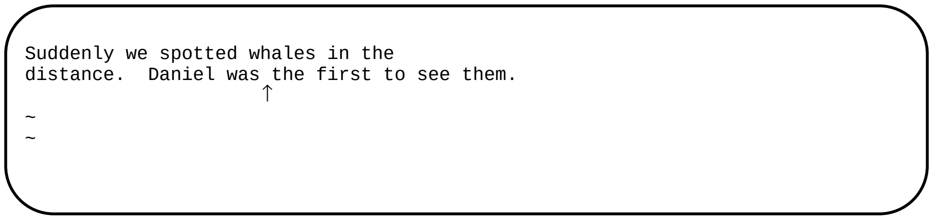
Positioning the Cursor by Sentences

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 spaces for **vi** to recognize them. You should get used to the convention of typing two spaces 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.

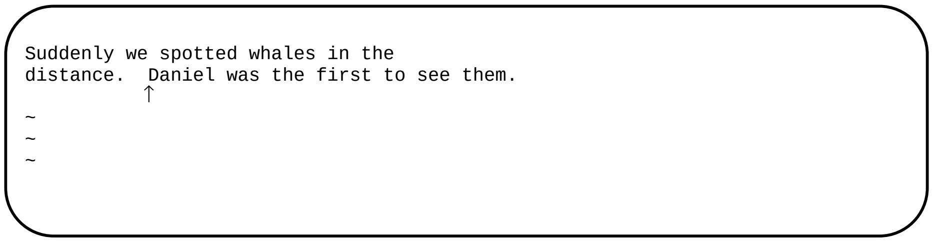
(	Move the cursor to the beginning of the current sentence.
*n***(**	Move the cursor to the beginning of the *n*th sentence above the current sentence.
)	Move the cursor to the beginning of the next sentence.
*n***)**	Move the cursor to the beginning of the *n*th sentence following the current sentence.

The example in the following screens shows how the open parenthesis moves the cursor around the screen.



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

```
(
```



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

Now repeat the command, preceding it with a number. For example, type:

```
3(
```

or

```
5)
```

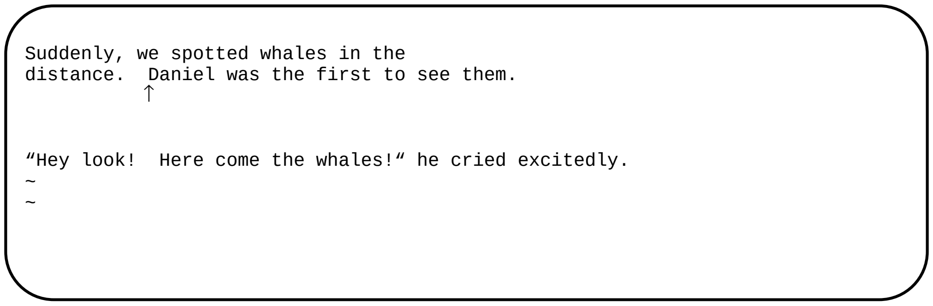
Did the cursor move the correct number of sentences?

Positioning the Cursor by Paragraphs

Paragraphs are recognized by **`vi`** if they begin after a blank line. If you want to be able to move the cursor to the beginning of a paragraph (or later in this tutorial, to delete or change a whole paragraph), then make sure each paragraph ends in a blank line.

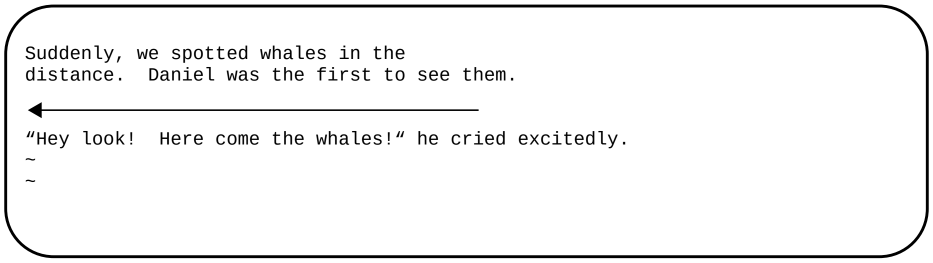
`{`	Move the cursor to the blank line beginning of the current paragraph, which is delimited by a blank line above it.
*n***`{`**	Move the cursor to the beginning of the *n*th paragraph above the current paragraph.
`}`	Move the cursor to the blank line ending this paragraph.
*n***`}`**	Move the cursor 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.



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


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

```
}
```



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

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

Positioning the Cursor in the Window

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

`H`	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.

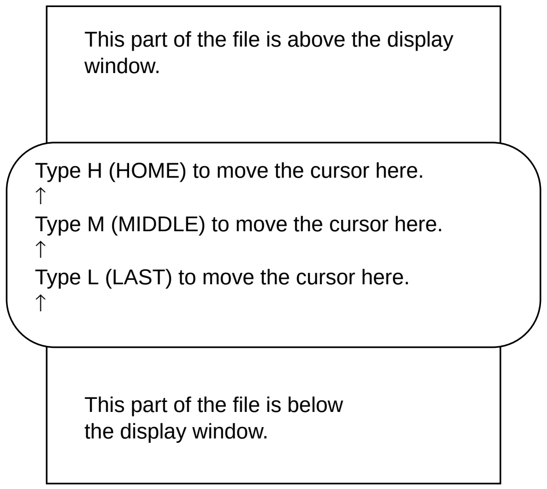

This part of the file is above the display window.
Type H (HOME) to move the cursor here.
↑
Type M (MIDDLE) to move the cursor here.
↑
Type L (LAST) to move the cursor here.
↑
This part of the file is below the display window.

Table 7-3 summarizes the **`vi`** commands for moving the cursor by positioning it on a character, line, word, sentence, paragraph, or position on the screen. Additional **`vi`** commands for moving the cursor are summarized in Table 7-9, which appears later in the chapter.

Table 7-3. Summary of vi Motion Commands

Positioning on a Character	
`h`	Move the cursor one character to the left.
`l`	Move the cursor one character to the right.
<BACKSPACE>	Move the cursor one character to the left.
<SPACEBAR>	Move the cursor one character to the right.
`f`*x*	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 the same direction on the line for the last character requested with **`f`**, or **`t`**. The **`;`** remembers the character and finds the next occurrence of it on the current line.
`,`	Continue searching in the opposite direction on the line for the last character requested with **`f`** or **`t`**. The **`,`** remembers the character and finds the next occurrence of it on the current 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).

Table 7-3. Summary of vi Motion Commands (Cont.)

Positioning on a Character	
-	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.

Table 7-4. Line Positioning Commands

Positioning on a Line	
$	Move the cursor to the last character on the line.
0 (zero)	Move the cursor to the first character on the line.
^ (circumflex)	Move the cursor to the first nonblank character on the line.

Table 7-5. Word Positioning Commands

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 white space.
b	Move the cursor backward one word to the first character of that word.
B	Move the cursor backward one word, which is delimited only by white space.
e	Move the cursor to the end of the current word.
E	Delimit the words by white space only. The cursor is placed on the last character before the next white space, or end of the line.

Table 7-6. Sentence Positioning Commands

Positioning on a Sentence	
(	Move the cursor to the beginning of the current sentence.
)	Move the cursor to the beginning of the next sentence.

Table 7-7. Paragraph Positioning Commands

Positioning on a Paragraph	
{	Move the cursor to the beginning of the current paragraph.
}	Move the cursor to the beginning of the next paragraph.

Table 7-8. Window Positioning Commands

	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.

Displaying Text Not Shown in the Current Editing Window

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 to another place in the file. This section covers those commands that can help you move around within the file by:

- scrolling forward or backward in the file
- moving to a specified line in the file
- searching for a pattern in the file.

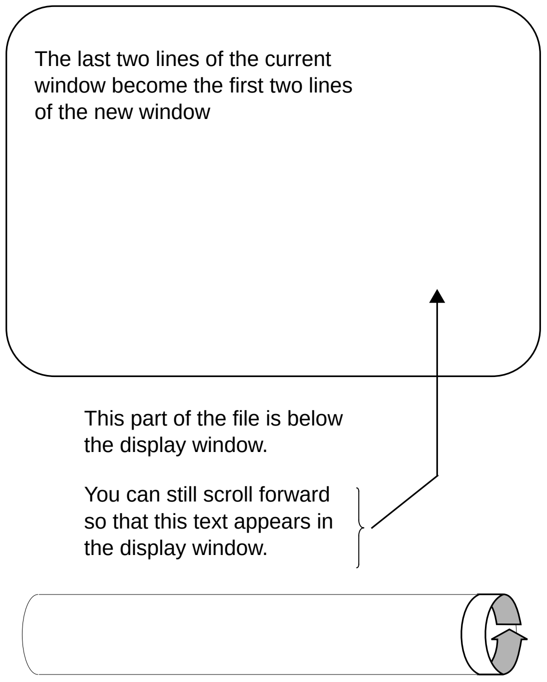
Scrolling Text

Four commands allow you to scroll the text of a file. The <`CTRL`><`f`> (control-f) and <`CTRL`><`d`> (control-d) commands scroll the screen forward. The <`CTRL`><`b`> (control-b) and <`CTRL`><`u`> (control-u) commands scroll the screen backward.

Scrolling Forward One Screen: The Control-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 two lines that were at the bottom of the current window are placed at the top of the new window. If too few lines are 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:

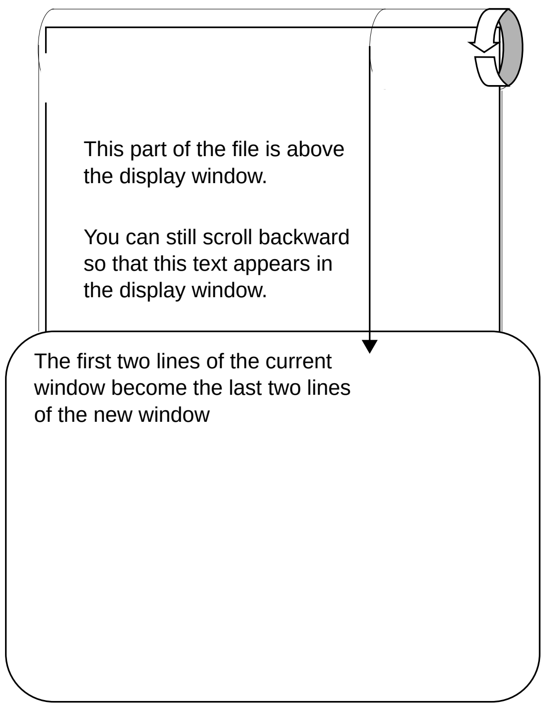

The last two lines of the current window become the first two lines of the new window
This part of the file is below the display window.
You can still scroll forward so that this text appears in the display window.

Scrolling Down a Half Screen: The Control-d Command

The <`CTRL`><`d`> command scrolls down a half screen to reveal text below the window. When you enter <`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 the cursor is on the last line of the file and you enter <`CTRL`><`d`>, a bell will sound.

Scrolling Back a Full Screen: The Control-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. The <`CTRL`><`b`> command leaves two reference lines from the previous window at the bottom of the screen. If not enough lines are above the current window to fill a full new window, a bell will sound and the current window will remain on the screen.


This part of the file is above the display window.
You can still scroll backward so that this text appears in the display window.
The first two lines of the current window become the last two lines of the new window

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.

Scrolling Back a Half Screen: The Control-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 up in the text, moving the portion above the screen into the window. Type:

```
<CTRL><u>
```

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

Moving to a Specified Line: The G Command

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, the **`G`** command sends the cursor 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.

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 the line 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
- whether the buffer has been modified since it was last written to a file
- the line number on which the cursor rests
- the total number of lines in the buffer
- the percentage of the total lines in the buffer represented by the current line.

```
This line is the 35th line of the buffer.
The cursor is on this line.
                    ↑


There are several more lines in the buffer.
The last line of the buffer is line 116.
~
~
```

<`CTRL`><`g`>

```
This line is the 35th line of the buffer.
The cursor is on this line.
                    ↑


There are several more lines in the
buffer.
The last line of the buffer is line 116.
~
~
“file.name” [modified] line 36 of 116 --34%--
```

Searching for Character Patterns: The / and ? Commands

The fastest way to reach a specific place in your text is to use 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 allow you to repeat the previous search.

The **`/`**, followed by a pattern (**`/`***pattern*), searches forward in the buffer for the next occurrence of the characters in the 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 the pattern, and puts the cursor on the first of those characters. For example, the command line

```
?data set design <RETURN>
```

finds the previous occurrence in the buffer of the words `data set design` and puts the cursor under the `d` in `data`.

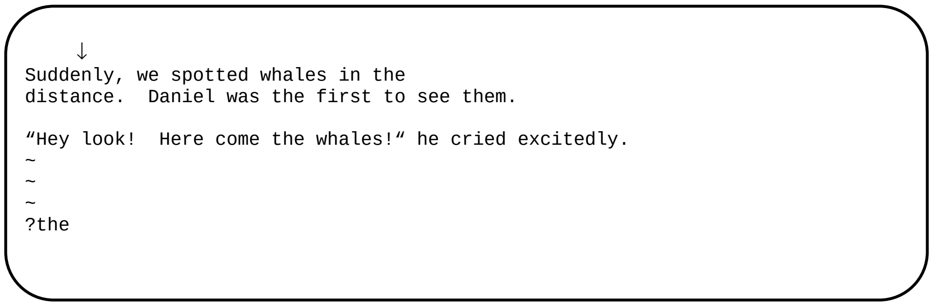
These search commands do not wrap around the end of a line while searching for two words. For example, you are searching for the words `Hello world`. If `Hello` is at the end of one line and `world` is at the beginning of the next, the search command will not find that occurrence of `Hello world`.

However, the search commands 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.

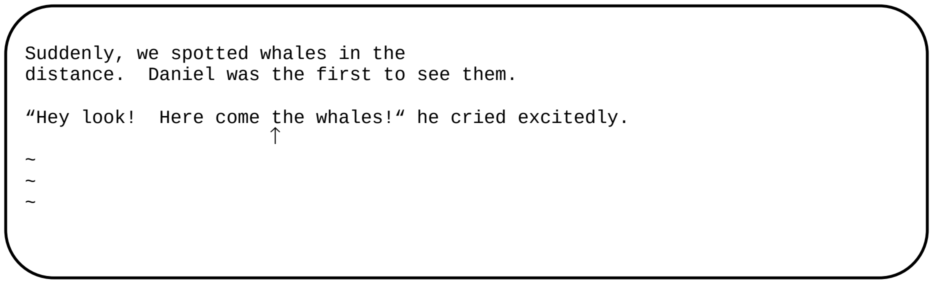
The **`n`** and **`N`** commands allow you to continue searches you have requested with **`/`***pattern* or **`?`***pattern* without retyping them.

`n`	Repeat the last search command.
`N`	Repeat the last search command in the opposite direction.

For example, 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`** command searches backward through the file and finds four occurrences of the character string `the`.

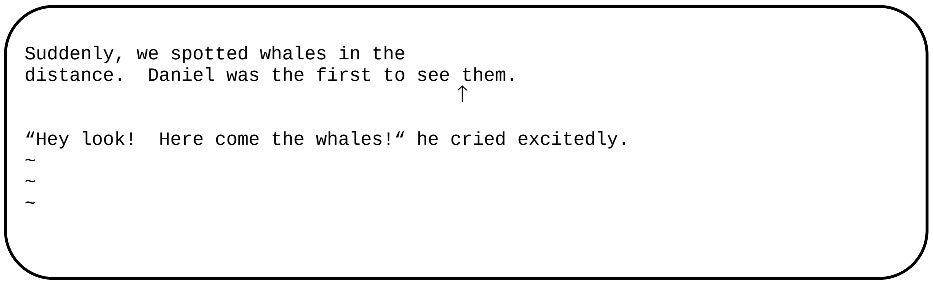


```
    ↓
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.

“Hey look!  Here come the whales!“ he cried excitedly.
~
~
~
?the
```



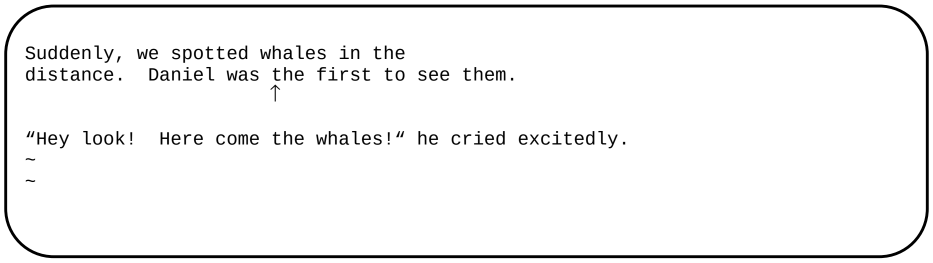
```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.

“Hey look!  Here come the whales!“ he cried excitedly.
                      ↑
~
~
~
```

n



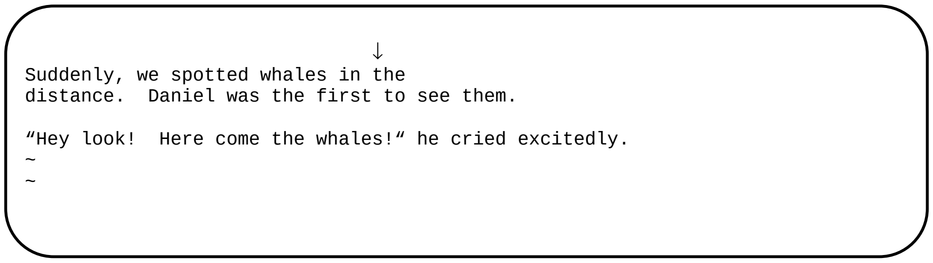
```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.
                                       ↑

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

n



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

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

n



```
                                ↓
Suddenly, we spotted whales in the
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.

Table 7-9 summarizes the **`vi`** commands for moving the cursor by scrolling the text, specifying a line number, and searching for a pattern.

Table 7-9. Summary of Additional vi Motion Commands

Scrolling	
Command	Function
`<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.	
Command	Function
`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	
Command	Function
`/`*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.

Positioning the Cursor and Scrolling with vi: Exercises

Exercise 2-1:

1. Create a file called **`exer2`**.
2. Type a number on each line, numbering the lines sequentially from 1 to 50.
   Your file should look similar to the following:

```
 1
 2
 3
 .
 .
 .

48
49
50
```

3. 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>`

4. Go to the end of the file.
5. Append the following line of text.

   `123456789 123456789`

6. What number does the command **`7h`** place the cursor on?
7. What number does the command **`3l`** place the cursor on?
8. Try the command **`$`** and the command **`0`** (zero).
9. Go to the first character on the line that is not a blank.
10. Move to the first character in the next word.
11. Move back to the first character of the word to the left.
12. Move to the end of the word.
13. Go to the first line of the file. Try the commands that place the cursor in the middle of the window, on the last line of the window, and on the first line of the window.
14. Search for the number 8. Find the next occurrence of the number 8. Find 48.

Positioning the Cursor and Scrolling with vi: Answers for Exercises

Answers for Exercise 2-1

Type:

```
vi exer2 <RETURN>
a1 <RETURN>
2 <RETURN>
3 <RETURN>
  .
  .
```

```
    .
48 <RETURN>
49 <RETURN>
50 <ESC>
```

Type:

```
<CTRL><f> <CTRL><b> <CTRL><u> <CTRL><d>
```

Notice the line numbers as the screen changes.

Type:

```
G
$
a<RETURN>
123456789 123456789 <ESC>
7h
3l
```

Typing `7h` puts the cursor on the `2` in the second set of numbers. Typing `3l` puts the cursor on the `5` in the second set of numbers.

`$` = end of line
`0` = first character in the line

Type:

```
^
w
b
e
```

Type:

```
1G
M
L
H
```

Type:

```
/8 <RETURN>
n
/48 <RETURN>
```

Creating Text with vi

Three basic commands enable you to create text:

a	Append text.
i	Insert text.

o Open a new line on which text can be entered.

After you finish creating text with any one of these commands, you can return to the command mode of **vi** by pressing the <ESC> key.

Appending Text with vi

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 Text with vi"* section. Make a new file named **junk2**. Append some text using the **a** command. To return to the command mode of **vi**, press the <ESC> key. Then compare the **a** command to the **A** command.

Inserting Text with vi

i Insert text before the cursor.

I Insert text at the beginning of the current line before the first character that is not a blank.

To return to the command mode of **vi**, press the <ESC> key.

In the following examples you can compare the append and insert commands. The arrows show the position of the cursor, where new text will be added.

- Example 1:

```
Append three spaces AFTER the H of Here.
                                   ↑
~
~
~
```

a <SPACEBAR> <SPACEBAR> <SPACEBAR>

```
Append three spaces AFTER the H of H   ere.
                                       ↑
~
~
~
```

- Example 2:

```
Insert three spaces BEFORE the H of Here.
                                    ↑
```

`i <SPACEBAR> <SPACEBAR> <SPACEBAR>`

```
Insert three spaces BEFORE the H of    Here.
                                      ↑
~
~
~
```

Notice that, in both cases, the user has left text insert mode by pressing the <ESC> key.

Opening a New Line with vi

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 line directly above or below the current line, and puts you into the insert 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, as shown in the following examples:

- Example 1:

```
Create text ABOVE the current line.
                  ↑
~
~
```

`O`

```
 ↓
 Create text ABOVE the current line.
~
~
```

- Example 2

```
Now create text BELOW the current line.
                   ↑
~
~
```

`o`

```
Now create text BELOW the current line.

↑
~
~
```

Table 7-10 summarizes the commands for creating and adding text with the **`vi`** editor.

Table 7-10. Summary of vi Create Text Commands

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 that is not a blank on the current line.
`o`	Create text at the beginning of a new line below the current line.
`O`	Create text at the beginning of a new line above the current line.
<ESC>	Return **`vi`** to command mode from insert mode.

Creating Text with vi: Exercises

Exercise 3-1:

1. Create a text file called **`exer3`**.
2. Insert the following four lines of text.

   ```
   Append text
   Insert text
   a computer's
   job is boring.
   ```

3. Add the following line of text above the last line:

   ```
   financial statement and
   ```

4. Using a text insert command, add the following line of text above the third line:

   ```
   Delete text
   ```

5. Add the following line of text below the current line:

   ```
   byte of the budget
   ```

6. Using an append command, add the following line of text below the last line:

   ```
   But, it is an exciting machine.
   ```

7. Move to the first line and add the word `some` before the word `text`.

   Practice using each of the six commands for creating text.

   Leave **`vi`** and go on to the next section to find out how to delete any mistakes you made in creating text.

Creating Text with vi: Answers for Exercises

Answers for Exercise 3-1:

1. Type:

   ```
   vi exer3 <RETURN>
   ```

2. Type:

   ```
   aAppend text <RETURN>
   Insert text <RETURN>
   a computer's <RETURN>
   job is boring. <ESC>
   ```

3. Type:

   ```
   financial statement and <ESC>
   ```

4. Type:

   ```
   3G
   iDelete text <RETURN>
   <ESC>
   ```

   The text in your file now reads:

   ```
   Append text
   Insert text
   Delete text
   a computer's
   financial statement and
   job is boring.
   ```

5. The current line is `a computer's`. To create a line of text below that line, use the **`o`** command.

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.`
   `<ESC>` leaves insert mode.

7. Type:

   ```
   1G
   /text
   i
   some<SPACEBAR> <ESC>
   ```

   `ZZ` writes the buffer to **`exer3`** and returns you to the shell.

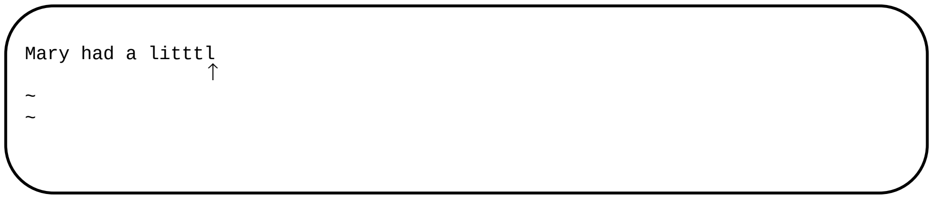
Deleting Text with vi

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

Deleting Text in Insert Mode

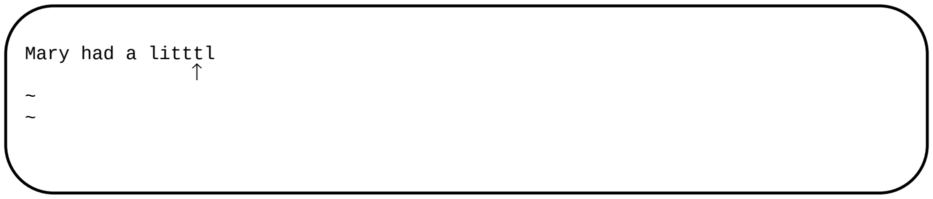
To delete a character at a time when you are in insert mode, use the `<BACKSPACE>` key; the cursor will move backward, silently “marking” each character it touches to be deleted. These characters are not actually erased from the screen, however, until you type over them or press the `<ESC>` key to return to the command mode.

In the following example, the arrows represent the cursor.



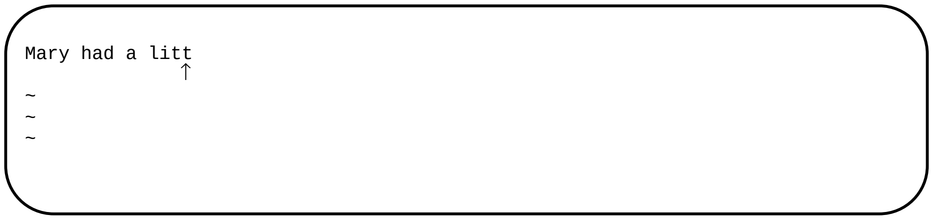
```
Mary had a litttl
               ↑
~
~
```

<BACKSPACE> <BACKSPACE>



```
Mary had a litttl
              ↑
~
~
```

<ESC>



```
Mary had a litt
             ↑
~
~
~
```

Notice that the characters were not erased from the screen until the <ESC> key was pressed.

Two other keys can also delete text in insert 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 they can be typed as text, precede the key with <CTRL><v>. Refer to the following table.

Table 7-11. CTRL-w, kill, and CTRL-v Editing Keys

<CTRL><w>	Undo the entry of the current word.
kill	Your terminal's “kill” character deletes all the text on the current line entered in insert mode. Type “stty” to display terminal settings. The @ is the default kill character.
<CTRL><v>	Remove the special meaning, if any, of the following character.

When you type <CTRL><w>, the cursor backs up over the word last typed and waits on the first character of that word. It does not erase the word from the screen until you press the <ESC> key or enter new characters over the old ones. The “kill” character behaves in a similar manner, except that it removes all text you have typed on the current line since you last entered the insert mode.

Undoing Changes in Command Mode: The u and U Commands

Before trying the delete commands, you should experiment with the undo command, **u**. This command undoes the last command you issued.

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, if 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.

Deleting Text 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 the command mode immediately remove unwanted text from the screen and, on most terminals, redraw the affected part of the screen.

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

[*number*] d*text_object*

Deleting Words: The dw Command

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

```
the deep dark depths of the lake.
    ↑
~
~
```

```
2dw
```

```
the depths of the lake.
    ↑
~
~
```

The **`dw`** command deletes one word or punctuation mark and the space(s) that follow it. You can delete several words or marks at once by specifying a number before the command. For example, to delete three words and two commas, type `5dw`.

```
the deep, deep, dark depths of the lake
    ↑
~
~
```

```
5dw
```

```
the depths of the lake
    ↑
~
~
~
```

Deleting Paragraphs: The d{ and d} Commands

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: The dd Command

To delete a line, type **dd**. To delete multiple lines, specify a number before the command. For example, typing

```
10dd
```

erases ten lines. If you delete more than five lines, **vi** displays the following notice on the bottom of the screen:

```
10 lines deleted
```

If fewer than ten lines are below the current line in the file, a bell sounds and no lines are deleted.

Deleting Text after the Cursor: The d and D Commands

To delete all text on a line after the cursor, put the cursor on the first character to be deleted and type:

`D` *or* `d$`

Neither of these commands allows you to specify a number of lines; they can be used only on the current line.

Table 7-12 summarizes the **vi** commands for deleting text.

Table 7-12. Summary of vi Delete Text Commands

For Command Mode:	
Command	Function
u	Undo the last command.
U	Restore current line to its previous state.
x	Delete the current character.
ndx	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 from the current position to the end of the current sentence.
d}	Delete from the current position to the end of the current paragraph

Table 7-12. Summary of vi Delete Text Commands (Cont.)

For Insert Mode:	
Command	Function
<BACKSPACE> <ESCAPE>	Delete the current character.
<CTRL><w> <ESCAPE>	Delete the current word.
@ (Kill Character) <ESCAPE>	Delete the current line of new text or delete all new text on the current line. (Note that the "kill Character" (i.e., @) is defined under stty in your .login file.)

Deleting Text with vi: Exercises

Exercise 4-1:

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.
   ```

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 insert mode.

   Delete the word `boring` while you are in command mode.

   What is another way you could have deleted the word `boring`?

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. Leave the screen editor and remove the empty file from your directory.

Deleting Text with vi: Answers for Exercises

Answers for Exercise 4-1:

1. Type:

   ```
   vi exer4 <RETURN>
   aWhen 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. <ESC>
   ```

2. Type:

   ```
   2G
   A
   , tedious, and unsavory <CTRL><w> <ESC>
   ```

   Press **`F`**`b` to get to the `b` of `boring`. Then type:

   `dw` *or* `6x`

3. You are at the second line. Type:

   ```
   2j
   I congenial and computerized <ESC>
   dd
   ```

   To delete the line and leave it blank, type in:

   `u` *(to undo the dd)*
   `0` *(zero moves the cursor to the beginning of the line)*
   `D`
   `d1G`

4. Write and quit **`vi`**:

   ```
   ZZ
   ```

   Remove the file:

   ```
   rm exer4 <RETURN>
   ```

Modifying Text with vi

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: The r and R Commands

r*x*	Replace the current character (the character shown by the cursor) with *x*. This command does not start insert mode, and so does not have to be followed by pressing the <`ESC`> key.
*n***r***x*	Replace *n* characters on the current line with *x*. This command automatically terminates after the *n*th character is replaced. It does not have to be followed by pressing the <`ESC`> key.
R	Replace only those characters typed over until the <`ESC`> command is given. If the end of the line is reached, this command appends the input as new text.

The **r** command replaces the current character with the next character 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:

```
rn
```

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 `7`.

```
The circus has 7777 ants.
```

Substituting Text: The s and S Commands

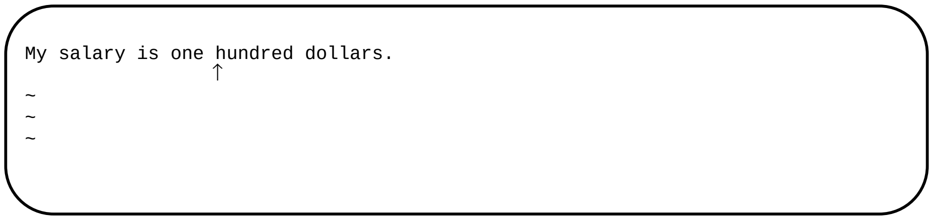
The substitute command replaces characters, but then allows you to continue to insert text from that point until you press the <`ESC`> key.

s	Delete the character shown by the cursor and append text. End insert mode by pressing the <`ESC`> key.
*n***s**	Delete *n* characters and append text. End insert mode by pressing the <`ESC`> key.
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 insert mode by pressing the <`ESC`> key.

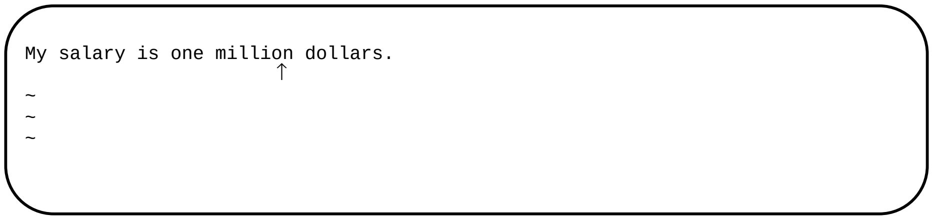
Notice that you cannot use a text-object argument with either **r** or **s**. Did you try?

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.



```
My salary is one hundred dollars.
                 ↑
~
~
~
```

```
7smillion <ESC>
```



```
My salary is one million dollars.
                       ↑
~
~
~
```

Changing Text: The c, cw, cc, and C Commands

The substitute command replaces characters. The change command replaces text-objects, and then continues to append text from that point until you press the `<ESC>` key. To end the change command, press the `<ESC>` key.

The change command can take a text-object argument. You can replace a character, word, an entire line, and so on, with new text.

*n***c***x*	Replace *n* text-objects of type *x*, such as sentences and paragraphs (shown by **)** and **}**, respectively).
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 command, **c**, uses a `$` sign to mark the last letter to be replaced. Notice how this works in the following example:

```
They are now due to arrive on Tuesday.
                              ↑
~
~
```

```
cw
```

```
They are now due to arrive on Tuesda$.
                              ↑
~
~
```

```
Wednesday <ESC>
```

```
They are now due to arrive on Wednesday.
                                      ↑
~
~
~
```

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

The **`C`** command, when used to change the remaining text on a line, works in the same way. When you enter the command it uses a `$` sign to mark the end of the text that will be deleted, puts you in insert mode, and waits for you to type new text over the old. The following screens show the use of the **`C`** command.

```
 This is line 1.
 Oh, I must have the wrong number.
 ↑
 This is line 3.
 This is line 4.
~
~
```

```
C
```

```
This is line 1.
Oh, I must have the wrong number$
↑
This is line 3.
This is line 4.
~
~
```

```
This is line 2. <ESC>
```

```
This is line 1.
This is line 2.
             ↑
This is line 3.
This is line 4.
~
~
```

Now try combining arguments. For example, type:

```
c{
```

Because you know the undo command, do not hesitate to experiment with different arguments or to precede the command with a number. You must press the <ESC> key before using the **u** command, because the **c** command places you in insert mode.

Compare the **S** and **cc** commands. Both produce the same results.

Table 7-13 summarizes the **vi** commands for changing text.

Table 7-13. Summary of vi Change Text Commands

Command	Function
r	Replace the current character.
R	Replace only those characters typed over with new characters until the <ESC> key is pressed.
s	Delete the character the cursor is on and append text. End insert mode by pressing the <ESC> key.
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.

Cutting and Pasting Text with vi

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.

Pasting Text: The p and P Commands

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 the 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 or below the current line.
P	Place the contents of the temporary buffer before the cursor or above the current line.

A partial line 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 the **p** key. The partial line is placed after the cursor.

Characters deleted by *n***x** also go into a temporary buffer. Any text object that was just deleted can be placed somewhere else in the text with the **p** command.

The **p** command should be used right after a delete command because the temporary buffer stores the results of only 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 later in this section.

Fixing Transposed Letters: The xp Command

A quick way to fix transposed letters is to combine the **x** and **p** commands as **xp**. The **x** deletes the letter, and the **p** places it after the next character.

Notice the error in the following 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: The y and yy Commands

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.

Table 7-14 summarizes the ways you can use the yank command.

Table 7-14. Summary of the Yank Command

Command	Function
*n***y***x*	Yank *n* number of text-objects of type *x* (such as sentences, shown by **)**, and paragraphs, shown by **}**).
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 the sentence.
y}	Yank all text up to the end of the paragraph.

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

Try the following command lines and see what happens on your screen. Remember, you can always undo your last command. Type:

```
5yw
```

Move the cursor to another spot. Type:

```
p
```

Now try yanking a paragraph (by typing **y}**) and placing it after the current paragraph. Then move to the end of the file (by typing **G**) and place that same paragraph at the end of the file.

Copying or Moving Text Using Registers

As you edit a file you may want to rearrange parts of your text or include the same text more than once in the file. **vi** provides a simple two-step procedure for moving or copying multiple lines of text: you store some text in a temporary file or "register," and then copy the contents of the register into the file being edited.

1. Store the text to be moved (or copied) in a register, using either of two commands: **y** (short for "yank") or **d** (short for "delete"). To do this, you must specify: (a) the number of text objects (such as lines or paragraphs) you want; (b) a double quote sign (″), the prefix for the name of a register; (c) a single-letter (lowercase) name (of your choice) for the register; (d) the appropriate command for extracting text (either **y** or **d**); and (e) the type of text object you want to extract. Enter the command as follows:

   [*number*][“*x*]*command*[*text_object*]

   The extracted text stays in the specified register until you either end the editing session or add more text (via yank or delete) to that register.

2. Copy the contents of the register into the file being edited with the **p** command. Enter the command as follows:

   [“*x*]p

For example, suppose you want to copy the first three lines of a file to the end of that file. You can do this by storing the three lines in a register called **a** and then copying **a** at the end of the file. Because you are copying (rather than moving) text, use the yank command, as follows:

```
3“ayy
```

Notice that the text object in this example is specified by a second **y** (after the **y** command). When the command character (either **y** or **d**) is duplicated in this way, the text object being requested is a line. Thus, in the above command line, you are yanking three lines and storing them in a register called **a**.

Now move the cursor to the end of the file. Type:

```
“ap
```

Did the lines you saved in register **a** appear at the end of the file?

Table 7-15 summarizes the cut and paste commands.

Cutting and Pasting Text with vi: Exercises

Exercise 5-1:

1. Enter **vi** with the file called **exer2** that you created earlier in this chapter.

   Go to line 8 and change its contents to `END OF FILE`.

Table 7-15. Summary of vi Cut and Paste Commands

Command	Function
p	Place the contents of the register containing the text obtained from the most recent delete or yank command into the text after the cursor.
yy	Yank a line of text and place it into a temporary buffer.
*n***y***x*	Yank a copy of *n* number of text objects of type *x* and place them in a temporary buffer.
″*r***y***x*	Place a copy of a text object of type *x* in the register *r*.
″*r***p**	Place the contents of the register *r* after the cursor.

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.

3. Go to line 8 and change its contents to `eight is great`.

4. Go to the last line of the file. Substitute `EXERCISE` for `FILE`. Replace `OF` with `TO`.

Cutting and Pasting Text with vi: Answers for Exercises

Exercise 5-1:

1. Type:

```
vi exer2 <RETURN>
8G
cc
END OF FILE <ESC>
```

2. Type:

```
1G
8“zyy
G
“zp
```

3. Type:

```
8G
cc
8 is great <ESC>
```

4. Type:

```
G
2w
cw
EXERCISE <ESC>
2b
cw
TO <ESC>
```

Using Other vi Commands

The following table contains some special commands you may find useful:

Table 7-16. Special vi Commands

.	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 . 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.” You want to correct your error, but you do not want to capitalize the `s` in the phrase “chemical states.” One way to solve 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 and J Commands

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 line that follows it.

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 the command mode. If you are in insert mode, press the <ESC> key to return to the command mode. Then type <CTRL><l> (control-l). **vi** erases the message and redraws the window exactly as it appeared before the message arrived.

Changing Lowercase to Uppercase and Vice Versa: The ~ Command

A quick way to change any lowercase letter to uppercase, or vice versa, is to put the cursor on the letter to be changed and type a **~** (tilde). For example, to change the letter `a` to `A`, press **~**. You can change several letters by typing **~** several times, or you can precede the command with a number to change several letters with that one command.

Table 7-17 summarizes the special commands.

Table 7-17. Summary of Special vi 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.

Using Line Editor (ex) 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 online *Command Reference.* This section discusses some of those most commonly used.

The **ex** commands are very similar to the **ed** commands discussed in the "Line Editor (ed) Tutorial" chapter.

Line editor commands begin with a **:** (colon). After you type the colon, the cursor drops to the bottom of the screen and displays the colon. The remainder of the command also appears at the bottom of the screen as you type it.

Returning to the Shell: The :sh and :! Commands

When you enter **vi**, the contents of the buffer fills 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 displays 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. These temporary escape commands are 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 are printed at the bottom of your screen. Type a shell command immediately after the `!`. The shell will run your command, give you output, and print the message `[Hit return to continue]`. When you press the <RETURN> key **vi** refreshes the screen and the cursor reappears exactly where you left it.

The **ex** command **:sh** allows you to do the same thing, but behaves differently on the screen. From the command mode of **vi** type **:sh** and press the <RETURN> key. A shell command prompt appears on the next line. Type your command(s) after the prompt, as you would normally do while working in the shell. When you are ready to return to **vi**, type <CTRL><d> or **exit**; your screen is refreshed with the contents of your buffer and the cursor appears where you left it.

Even changing directories while you are temporarily in the shell will not prevent you from returning to the **vi** buffer where you were editing your file when you type **exit** or <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`** is created. It will contain only the current line in the **`vi`** buffer.

You can also write a whole section of the buffer to a new file by specifying a range of lines. For example, to write lines 23 through 37 to a file, type the following:

`:23,37w` *newfile* `<RETURN>`

Moving to a Specified Line: The : Command

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: The :.,$d Command

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>
```

- **`.`** (period) represents the current line
- `$` (dollar sign) represents the last line.

Reading a File into the Buffer: The :r Command

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: The :g Command

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 a 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 the words double helix. The **`ex`** editor's global command allows you to do this with one command line. First, you have 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/p<RETURN>.
    The line editor prints 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 changes every occurrence of pattern
    on the current line.
    If you do not add the letter g,
    ex changes only the first occurrence.
```

`:g/helix/s//double helix/g` <RETURN>

```
    This command line searches for the word helix.
    Each time the word helix is found,
    the substitute command substitutes two words,
    double helix, for every instance of
    helix on that line.
    The delimiters after the s command do not require
```

```
that the word helix be 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 the ex(1) page in the online Command Reference
and Chapter 6, “Line Editor (ed) Tutorial”.
```

Table 7-18 summarizes the **ex** line editor commands used with **vi**.

Table 7-18. Summary of Using ex Line Editor Commands with vi

Command	Function
:	Indicate that the commands following are **ex** commands.
:sh <RETURN>	Temporarily return to the shell to perform shell commands.
<CTRL><d>	Escape the temporary shell and return to the current window of **vi** to continue editing.
:*n* <RETURN>	Go to the *n*th line of the buffer.
:*x,y*w *file* <RETURN>	Write lines from the number *x* through the number *y* into a new file (*file*).
:$ <RETURN>	Go to the last line of the buffer.
:.,$d <RETURN>	Delete all the lines in the buffer from the current line to the last line.
:r *shell.file* <RETURN>	Insert the contents of *shell.file* after the current line of the buffer.
:s/*text*/*new_words*/ <RETURN>	Replace the first instance of the characters *text* on the current line with *new_words*.
:s/*text*/*new_words*/g <RETURN>	Replace every occurrence of *text* on the current line with *new_words*.
:g/*text*/s//*new_words*/g <RETURN>	Replace every occurrence of *text* in the file with *new_words*.

Quitting vi

Five basic command sequences can be used to quit the **vi** editor. Commands that are preceded by a colon (:) are **ex** commands. (Refer to the following table).

Table 7-19. Commands to Quit the vi Editor

:wq <RETURN>	Write the contents of the **vi** buffer to the UNIX file currently being edited and quit **vi**.
ZZ	Write the buffer only if the contents of the buffer changed since the last write command 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** warns you that you must either save the buffer or use the **:q!** <RETURN> command to terminate.

The **:wq** command and the **ZZ** command, under the circumstances explained above, 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 contents of the buffer back into the file. Type:

```
:wq <RETURN>
```

The **vi** editor 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 responds with:

```
“johnson” File exists - use “w! johnson” to overwrite
```

If you want to replace the contents of the existing file with the contents of the buffer, use the **`:w!`** command to overwrite **`johnson`**.

```
:w! johnson <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, leave **`vi`** without writing to the file. Type:

```
:q! <RETURN>
```

Table 7-20 summarizes the quit commands.

Table 7-20. Summary of vi Quit Commands

Command	Function
`ZZ`	Write the file if it has changed since the last write command 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 the buffer to a file even if the buffer changed.
`:q` <RETURN>	Quit **`vi`** without writing the buffer to a file only if the buffer has not changed.

Using vi Command Line Options

Several command line options are available when invoking **`vi`**. These options allow you to:

- recover a file lost if **`vi`** is interrupted
- place several files in the editing buffer and edit each in sequence
- view a file without risk of accidentally changing the file.

Recovering a Lost File: The `-r` Option

If an interrupt or disconnect occurs, **`vi`** exits without writing the text in the buffer back to its file. However, **`vi`** stores a copy of the buffer for you. When you log back in to the

UNIX system, you will receive a system message that indicates that you can recover the file with the **-r** option of the **vi** command.

Type:

```
vi -r filename <RETURN>
```

All or most of the changes you made to *filename* before the interrupt occurred are now in the **vi** buffer. You can continue editing the file, or you can write the file and quit **vi**. The **vi** editor will remember the filename 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 filename. 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 is a message at the bottom of the screen giving the name of the file and the number of lines and characters in that file. Then you can edit the next file 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: Using view

It is often convenient to be able to inspect a file by using the powerful search and scroll capabilities of **vi**. However, you might want to protect yourself against accidentally changing a file during an editing session. The read-only option prevents you from writing in a file. To avoid accidental changes, you can set this option by invoking the editor as **view** rather than **vi**.

Table 7-21 summarizes the command line options for **vi**.

Table 7-21. Summary of vi Command Line Options

Option	Function
`vi` *file1 file2 file3* <RETURN>	Enter three files (*file1*, *file2*, and *file3*) into the **`vi`** buffer to be edited.
`:w` <RETURN> **`:n`** <RETURN>	Write the current file and start editing the next file.
`vi -r` *file1* <RETURN>	Recover the changes made to *file1*.
`view` *file* <RETURN>	Inspect file with the read-only option set, preventing accidental changes to *file*.

Using vi Command Line Options: Exercises

Exercise 6-1:

1. Try to restore a file lost by an interrupt.

   Enter **`vi`**, create some text in a file called **`exer6`**.
2. Turn off your terminal without writing to a file or leaving **`vi`**.
3. Turn your terminal back on, and log in again.
4. Try to get back into **`vi`** and edit **`exer6`**.

Exercise 6-2:

1. Place **`exer1`** and **`exer2`** in the **`vi`** buffer to be edited.
2. Write **`exer1`** and call in the next file in the buffer, **`exer2`**.
3. Write **`exer2`** to a file called **`junk`**.
4. Quit **`vi`**.

Exercise 6-3:

1. Try out the command:

   ```
   vi exer* <RETURN>
   ```

   What happens?
2. Try to quit all the files as quickly as possible.

Exercise 6-4:

1. Look at **`exer4`** in read-only mode.
2. Scroll forward.
3. Scroll down.

4. Scroll backward.
5. Scroll up.
6. Quit and return to the shell.

Using vi Command Line Options: Answers for Exercises

Answers for Exercise 6-1:

Type:

```
vi exer6 <RETURN>
a(append several lines of text)  <RETURN>
```

Turn off the terminal.

Turn on the terminal.

Log in on your UNIX system.

Type:

```
vi -r exer6 <RETURN>
:wq <RETURN>
```

Answers for Exercise 6-2:

Type:

```
vi exer1 exer2 <RETURN>
:w <RETURN>
:n <RETURN>
:w junk <RETURN>
ZZ
```

Answers for Exercise 6-3:

Type:

```
vi exer* <RETURN>
```

`vi` calls all files with names that begin with **`exer`**, so it displays the following message:

```
8 files to edit
```

Type

```
:q!
```

Answers for Exercise 6-4:

Type:

```
view exer4 <RETURN>
<CTRL><f> <CTRL><d> <CTRL><b> <CTRL><u> :q <RETURN>
```

Displaying and Setting Environment Options

The **ex** line editor provides several options to help you customize your **vi** editing environment. These options are listed in Table 7-22.

NOTE

This list may not contain all the options available for your system.

Table 7-22. Summary of vi Environment Options

Option Name	Abbreviation	Default
autoindent	ai	noai
autoprint	ap	ap
autowrite	aw	noaw
beautify	bf	nobf
directory	dir	dir=/tmp
edcompatible	ed	noed
errorbells	eb	noeb
exrc	ex	noex
flash	fl	fl
hardtabs	ht	ht=8
ignorecase	ic	noic
lisp	—	nolisp
list	—	nolist
magic	—	magic
mesg	—	mesg
modelines	ml	noml
number	nu	nonu
novice	—	nonovice
optimize	opt	noopt

Table 7-22. Summary of vi Environment Options (Cont.)

Option Name	Abbreviation	Default
paragraphs	para	para=IPLPPQPP LIpplpipnpb
prompt	—	prompt
readonly	ro	noro
redraw	—	noredraw
remap	—	remap
report	—	report=5
scroll	scr	scr=12
sections	sect	sect=NHSHH HUuhsh+c
shell	sh	sh=*yourloginshell* (e.g. `ksh`, `sh`)
shiftwidth	sw	sw=8
showmatch	sm	nosm
showmode	smd	nosmd
slowopen	slow	noslow
tabstop	ts	ts=8
taglength	tl	tl=0
tags	—	tabs=tags /usr/lib/tags
term		term=`TERM`*value* (e.g. `630`, `vt100`)
terse	—	noterse
timeout	to	timeout
ttytype	tty	tty=*machinedependent*
warn	—	warn
window	wi	wi=24
wrapmargin	wm	wm=10
wrapscan	ws	ws
writeany	wa	nowa

As you can see from Table 7-22, there are three types of options:

on/off — these options are turned off by typing `no` in front of the option name as shown in the following example for the `ignorecase (ic)` option:

- `ic` ignores uppercase and lowercase distinctions
- `noic` recognizes upper and lowercase distinctions.

numeric — these options require a numeric value, as shown in the following examples for the `wrapmargin(wm)` option:

- `wm=10` inserts a `<RETURN>` (carriage return) on or before ten spaces from the left of the right edge of the screen.
- `wm=8` inserts a `<RETURN>` (carriage return) on or before eight spaces from the left of the right edge of the screen.

string these options require a value as shown in the following examples for the `term` and `shell (sh)` options:

- `term=xterm`
- `sh=`**`/bin/csh`**

Frequently Used ex Options

This section describes the following options for customizing your **`vi`** environment:

- `autowrite`
- `ignorecase`
- `list`
- `number`
- `term`
- `wrapmargin`
- `wrapscan`

The autowrite Option

If the `autowrite` (`aw`) option is on, a **`:w`** command will automatically be performed whenever you temporarily leave the file, or change the file with an **`ex`** command (such as **`:n`**, **`:!`**, **`:sh`**, **`:tag`**, and **`:rew`**).

Any changes that you made to a file will automatically be written to the file.

- to turn on this option with the **`:set`** command, type

  ```
  :set aw <RETURN>
  ```

- to turn off this option with the **`:set`** command, type

  ```
  :set noaw <RETURN>
  ```

The ignorecase Option

The `ignorecase` (`ic`) option is very useful for searches. When turned on, this option makes no distinctions between uppercase and lowercase letters during searches. When turned off, it distinguishes between uppercase and lowercase. For example, if you type

```
/REturn
```

- to find all instances of `Return`, `return`, and `<RETURN>`, use the following **`:set`** command

  ```
  :set ic <RETURN>
  ```

- to find all instances of `REturn` in the text to correct a typographical error, use the following **`:set`** command

  ```
  :set noic <RETURN>
  ```

The list Option

The `list` option is very useful for finding characters that don't normally print on the screen. When turned on, tab characters are marked by a `^I` character and the end of each line is marked by a `$` character. For example,

- when this option is turned on,

  ```
  :set list <RETURN>
  ```

  you can see the hidden space and tab characters and the end of the line is marked:

  ```
  chicks^Iducks^Igeese^Ibetter scurry                $
  ```

- when this option is turned off,

  ```
  :set nolist <RETURN>
  ```

  the same line looks like this

  ```
  chicks     ducks    geese     better scurry
  ```

This is option particularly useful for `troff` (text formatting tool) tables.

The number Option

The `number` option is used to display line numbers in a file. This is useful for **`vi`** commands that use line addressing such as the **`G`** command.

- to turn on this option with the **`:set`** command, type

  ```
  :set nu <RETURN>
  ```

- to turn off this option with the **`:set`** command, type

  ```
  :set nonu <RETURN>
  ```

The term Option

Occasionally, you may find **`vi`** operating oddly. Often, this is due to your terminal setting being incorrectly defined. You can use the `term` option to set the correct terminal type. To do this with the **`:set`** command, type

```
:set term=terminal_type <RETURN>
```

The wrapmargin Option

If you want your text to be a certain width on the screen, you can use the `wrapmargin` (`wm`) option to set the right margin. To do this with the **`:set`** command type

```
:set wm=n <RETURN>
```

n represents the number of characters from the right side of the screen where you want an automatic carriage return to occur. **`vi`** automatically breaks lines by inserting a `<RETURN>` character between words. An automatic return will be entered only between words, not between syllables of a word.

The `<RETURN>` is inserted as close as possible to the new right margin. For example, if you want a carriage return at ten characters from the right side of the screen, type:

```
:set wm=10 <RETURN>
```

The new right margin will be ten spaces to the left of the right edge of the screen.

The wrapscan Option

The `wrapscan` option is useful for searches. When this option is turned off, searches stop at the end of a file. When this option is turned on, searches continue from the start of the search, through the end of the file, and back through the beginning of the file to the start of the search. With `wrapscan` turned on, searches are continuous around the end (or beginning) of the buffer.

- to turn on this option with the **`:set`** command, type

  ```
  :set ws <RETURN>
  ```

- to turn off this option with the **`:set`** command, type

  ```
  :set nows <RETURN>
  ```

Setting and Displaying Options During a vi Session: The :set Command

To display current option settings, when you are in **`vi`**:

- type the following command to display the settings of all options you changed
  - during an editing session
  - with the `EXINIT` environment variable
  - in your **`.profile`** file

  ```
  :set <RETURN>
  ```

- type the following command to display the settings of all options

  ```
  :set all <RETURN>
  ```

You can also use the **`:set`** command to create or change the environment options during single editing session, as shown below:

:set *option*
:set *nooption*
:set *option=number*
:set *option=string*

For example, if you are working in **`vi`** want to distinguish between uppercase and lowercase letters during a search, type

```
:set noic <RETURN>
```

This will set the `ignorecase` option to `noignorecase` for a single **`vi`** session. If you leave **`vi`** and then return to **`vi`**, the options will return to the defaults that are set by the `EXINIT` environment variable discussed later in this section.

Setting Options for a Single Login Session

You can set environment options for a single login session by setting and exporting the `EXINIT` environment variable at the shell prompt. Separate each environment option by a space and enclose them in quotes as shown in the following example:

```
$ EXINIT="set ic wm=10 sh=/bin/ksh"
$ export EXINIT
```

Setting vi Environment Options for All Login Sessions

If you want your options to be set automatically every time you use **`vi`**, you can

- set the `EXINIT` environment variable in your **`.profile`** file
- list **`:set`** commands in a file called **`.exrc`** in your home directory.

NOTE

The `EXINIT` environment variable overrides the options set in a **`.exrc`** file. If the `EXINIT` environment variable is present, the UNIX system will not check for a **`.exrc`** file.

Defining EXINIT in Your .profile

If you want to set your environment options for all your login session, edit your **`.profile`** and include and export the `EXINIT` variable as shown in the following example:

```
EXINIT="set ic wm=10 sh=/bin/ksh"
export EXINIT
```

Creating a .exrc File

You can use a **.exrc** file to set options.

- a **.exrc** file in your home directory will set options for all your files in all your directories .
- a **.exrc** file in subdirectory will only set options for files in that directory.

To create a **.exrc** file

1. Enter an editor with a **.exrc** filename:

   ```
   $ vi .exrc <RETURN>
   ```

2. Type one line of text for each option as shown in the following example:

   ```
   :set wm=10 <RETURN>
   :set ic <RETURN>
   :set sh=/bin/ksh <RETURN>
   ```

3. Finally, write the contents of the buffer to the file and quit the editor.
4. The next time you use **vi**, **.exrc** the options you set will be used.

NOTE

The options set using the EXINIT environment variable override options set in the **.exrc** file. If EXINIT is present, **.exrc** options are not used.

EXINIT and **.exrc** may include other **ex** commands such as **:map** and **:abbrev;** see the **ex(1)** page in the online *Command Reference.*

8 LP Print Service Tutorial

8
LP Print Service Tutorial

Introduction

The LP print service is a set of UNIX system programs that help you print files on paper. The name "LP" stands for "line printer," the type of printing device for which the print.service was designed originally. Now, however, because the print service can accommodate many types of printing devices, the name "LP" is more historical than descriptive.

The simplest way to use this print service is by running the **lp** command and specifying the name of the file you want to print. The **lp** command routes a job request to a destination (such as a line printer or a laser printer) where it is placed in a queue to await printing. The destination may be a printer or a class of printers. If you don't specify a destination, the request is routed to the default destination.

When you enter such a command line, the system responds with a message that (a) confirms the name of the printer doing the job, (b) assigns an ID number to your print request, and (c) acknowledges the number of files you've asked to have printed.

```
$ lp filename
    request id is laser-9885 (1 file)
```

The system response shows that your job will be printed on a printer named `laser` (the default printer for your system) and consists of one file. The string (set of characters) `laser-9885` is called a "request-ID"; use it to refer to a job when you are checking its status.

If you print a file with this simple command, you don't need to make decisions about issues such as the size of the paper; you can assume that when your system administrator set up the print service he or she chose default values for specifications such as paper size. (If you are the administrator of the print service for your computer, see the "*LP Print Service Administration*" chapter in *System Administration.)* You are not limited to the use of these default values, however. The following section explains how you can customize your print job.

Providing Your Own Print Specifications

The LP print service allows you to provide your own specifications for many aspects of a print job.

- Do you want your file to be printed by a particular printer or a particular type of printer?

Default: assigned by the print service administrator.

- Do you want to print your file on plain paper (the stock selected by the administrator)? Or do you want to use pre-printed forms, such as invoices or invitations?

  Default: plain paper.

- Do you want a particular font?

  Default: assigned by the print service administrator.

- Do you want pages of a particular size?

  Default: assigned by the print service administrator.

- Do you want to increase or decrease the number of lines of text that appear in each inch on the page?

  Default: assigned by the print service administrator.

  Do you want a "banner page" to be printed along with your file? If the Enhanced Security Utilities are installed, a banner page is always printed, even if you specify otherwise.

- Do you want the security level of the data printed on every page of the output?

  Default: the security level is printed.

- If you're printing more than one file, do you want them to be printed as one job (so there are no page breaks to mark separate files) or do you want them to be printed in discrete segments, so each file begins on a new page?

  Default: files are printed separately.

- Do you want more than one copy?

  Default: one copy is printed.

If you specify only a filename when you run the **`lp`** command (as discussed above), you do not need to answer any of these questions; your administrator will have answered them when setting up the print service on your system. If you want "non-default" specifications for your job, however, you will have to provide them on the command line.

There are several options you can include on your command line that will let you provide your own job specifications. To understand these options, a brief look at the main components of a print job will help you.

Components of the LP Printing Process

Printing a document with the **`lp`** command requires the interaction of five key components: (1) your electronic file, (2) the **`lp`** program, (3) any filters your administrator has installed (and you have requested), (4) any character sets (CS) or print wheels your administrator has installed (and you have requested), and (5) the paper on which your file is printed. The role of each component in the printing process is summarized in Figure 8-1.

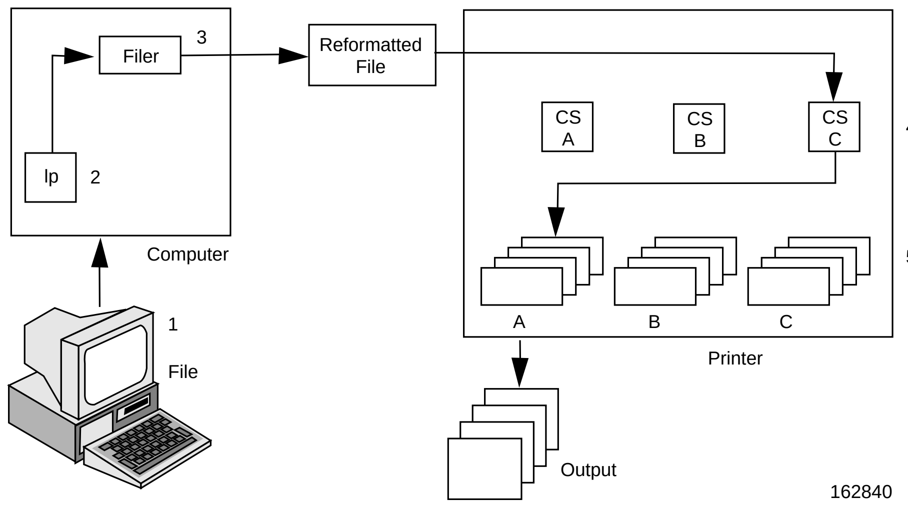


Figure 8-1. Main Components of a Print Job

In the example shown in Figure 8-1, the person printing the file has selected type C character set and type A forms. Both of these selections are made when you request a print job.

LP Security

NOTE

This section assumes you understand Mandatory Access Control (MAC) and security levels. If you do not, read the *"Mandatory Access Control"* section of Chapter 14, "*Managing Files Securely*" of this guide before reading this section.

All items that contain data (such as files or pipes) have a security level associated with them. These security levels reflect the sensitivity of the data; a higher security level means the data is more sensitive to disclosure. Each user of the system also has a security level. The computer system compares the user's security level with the security level of the data to determine if the user can access the data.

The LP system uses security level checks to determine if you can print a file. You are allowed to print a file only if you have MAC read access to it.

Once a file is printed, the computer system can no longer maintain access control of the data. Unless the printed copy indicates the sensitivity of the data, access to the data cannot be controlled properly. Therefore, the LP system automatically prints security level information on each page of paginated output and on the banner and trailer pages for each job. (A banner page is a page the system prints to mark the beginning of a print job; the trailer page marks the end.)

You can override the printing of security level information on the printed output, but not on the banner and trailer pages. In addition, it is not possible to cancel the banner and trailer pages. Thus, every print job always has complete security level information associated with it.

Each printer on your system has a security level range associated with it. The printer can print a file only if the file's security level is within the security level range of the device. The security level range of the printer should reflect the physical security of the printer itself. For example, if the printer is in an open location, it should not be used to print sensitive information.

About This Chapter

This chapter describes three functions you may do while printing files:

- enabling and disabling a printer.
- controlling the process of printing by selecting a printer, monitoring the printing process through messages and status reports, and, when necessary, changing and canceling requests.
- controlling the appearance of the finished document by providing print specifications such as fonts and the page size.

Enabling and Disabling a Printer

NOTE

The **enable** and **disable** commands are not always available to users. Because enabling and disabling printers is an administrative function, it is left to the discretion of the system administrator to decide who should have access to these commands. If you have access to these commands, you can enable or disable a printer only if your security level is within the range of the printer.

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

```
$ enable printer_1 printer_2 printer_3 <RETURN>
```

You can verify that you have enabled a printer by requesting a status report for it (see "*Requesting Status Reports on Printers*" later in this chapter).

There may be times when you want a printer to stop printing jobs. For example, hardware malfunctions, paper jams, running out of paper, and shutdowns at the end of the day are all situations that may require stopping the printer. To stop printing, deactivate the printer by issuing the **`disable`** command.

```
$ disable printer_1 <RETURN>
```

The printer will stop printing the current job and save it to complete later (when the printer has been enabled again).

There are other ways to have the current job handled, however. You may have the current job completed immediately, before the printer is disabled, by using the **`-W`** option.

On the other hand, you may not care whether or not it is completed at all. For example, if the output being produced is full of printing errors (such as parts of the text being illegible because of a lack of toner in the printer), you'll have to start the job over from the beginning anyway, once you have resolved the problem. In cases such as this, you'll want to disable the printer and cancel the job at the same time. To do so, specify the **`-c`** option; the job currently being printed will be thrown out as the printer is disabled. The **`-W`** and **`-c`** options are mutually exclusive.

NOTE

The **`-c`** and **`-W`** options will not work for jobs being sent to a printer on a server system. This is because the **`enable`** and **`disable`** commands do not actually activate or deactivate printers on server systems; instead, they activate or deactivate the transfer of files to a server system. As a result, the **`-c`** and **`-W`** options are ignored when requested with the **`disable`** command for a server printer.

Finally, when you disable a printer, it is a good idea to record the reason for your action so other users may understand why a particular printer is unavailable. To record your reason, add the **`-r`** option, followed by a reason, to the command line. Be sure to enclose your reason in double quotes, so it will be treated as a single argument.

```
$ disable -r “disabling for reconfiguration” printer4
<RETURN>
```

The reason you provide will be displayed by the **`lpstat`** command when a user requests a status report on that printer. For example, if you specify `paper jam` as the reason when you disable printer `printer_1`, a user who later runs **`lpstat`** to determine the status of `printer_1` will receive the response shown below:

```
$ lpstat -p printer_1
    printer printer_1 disabled since July 18 10:15-
    paper jam
$
```

If you disable a printer without supplying a reason, subsequent output from the **`disable`** command will include the message `unknown reason.`

Controlling the Printing Process

The LP print service allows you to monitor and control the printing process by doing the following:

- specifying a printer for your job
- identifying high priority jobs that need to be “pushed to the front” of the job queue
- requesting status reports about printers, print service resources (such as forms and character sets), and jobs in progress
- changing the specifications of a job request already submitted
- canceling a job in progress.

This section explains how to do these tasks.

Selecting a Print Destination

The term “print destination” refers to any device your system administrator has defined to be a printer or a class of printers. A class is a set of printers grouped together by an administrator for convenience. For example, one administrator might group together all printers of a similar type (such as laser printers or line printers) into a class. Another administrator might assign all the printers on the second floor of a building to the same class. A class can be defined in any way that is convenient for the administrator and/or the users of a print service; the LP print service does not require a printer to meet any prerequisites before it is assigned to a class. In this sense, a class is an arbitrary grouping.

The **`-d`** *dest* (short for destination) option on the command line causes your file to be printed at the destination specified in the *dest* argument, as long as a printer is available to you and capable of meeting your specifications for the job. (Although a printer may be reported by **`lpstat`** as being available, whether or not it's available to you depends on your level and the level range of the printer.) In the following example, a request is made to have a file called **`memo`** printed on `printer_3`.

```
$ lp -d printer_3 memo <RETURN>
```

Security Considerations

NOTE

This section assumes you understand Mandatory Access Control (MAC) and security levels. If you are not familiar with MAC and security levels, see the *“Mandatory Access Control”* section of Chapter 14, “*Managing Files Securely*” of this guide before reading this section.

On your system, each printer will have a range of security levels associated with it. This security level range is intended to keep sensitive information safe and is based on the physical security of the printer. If a printer is accessible to everyone, it should obviously not be used to print sensitive material, because someone could easily see or acquire the material. When configuring the printer system, your system administrator will have assigned appropriate security levels to each printer.

If you submit a print job that is not within the printer's security level range, the LP system rejects the job. In that case, you should choose another printer as the destination for your print job. Your system administrator can give you information on the level ranges assigned to your site's printers.

Using a Server Printer

NOTE

If your system is being run in compliance with B1- or B2-level security, the following does not apply; all printers in a B2 operating environment must be directly connected.

The LP print service allows computers connected through a network to share printing tasks. If the printer you choose is connected to a server computer, there may be some delays in having files printed, and in receiving responses to queries about job and printer status from a server machine. Otherwise, however, the location of a printer to which you are sending print requests and related commands should not be obvious to you: follow the same procedures every time you use the print service, regardless of whether the printer being used is a client or a server.

Controlling Priorities in the Job Queue

As you and other users send requests for print jobs to the printers on your system, your requests are arranged in a queue that determines the order of printing. Highest priority is given to requests that have been assigned level 0 priority; lowest priority to requests with a level of 39. Whether your job is assigned high or low priority depends on several factors.

First, the default value for job priority on your system is 20, unless your system administrator has defined it otherwise. Every job you submit to a printer will be given this medium-level priority. If your administrator has redefined the default priority level so that it is now, for example, 10, all jobs that you send to the printer will be given this higher priority.

You can change this priority level, however, by requesting a level other than the default; to do so, use the **`-q`** option to the **`lp`** command. For example, if you need a memo printed immediately, you can send it to the front of the queue by assigning it the highest priority: `0`.

```
$ lp -d printer_3 -q 0 urgent.memo <RETURN>
```

Note that the system administrator can limit the priority level you can use. If your administrator has limited the priority level available to you and you request a priority higher than that, the priority level will remain, by default, at the level set by the administrator. Check with your system administrator to find out what the default priority level is and whether there is a limit on the priority level you can request.

Requesting Messages from the Print Service

The LP print service does not automatically notify you when your job has been printed. To make sure you will be notified, list the **-w** option on the **lp** command line, as follows:

$ **lp -d printer_3 -w** *filename* <RETURN>

The print service will display a message on your terminal screen to let you know when your file has been printed. If you are not logged in when the message is ready to be sent, the message will be sent to you via electronic mail instead.

If you want to be notified through electronic mail that your file has been printed, include the **-m** option after the **lp** command, as follows:

$ **lp -d printer_3 -m** *filename* <RETURN>

Requesting Status Reports on Printers

At some time after issuing a request for a print job, you may want to find out whether it is proceeding properly or if problems have arisen. You can check the status of all print requests by executing the **lpstat** command. When issued alone (without any options), this command will tell you the status of all requests you have made to the LP print service, as shown in the following example:

```
$lpstat
printer_1-25pr2cms1942July 19 13:09
printer_1-26pr2cms3893July 19 13:15
printer_1-27pr2cms 942July 19 14:09
```

If you do not want to know about all print requests, you can specify a subset of requests by listing the request ID numbers for those jobs on the command line. (Whenever a print request is issued, a request ID number for it is displayed on the screen.)

$ **lpstat laser-6885 printer_1** <RETURN>

In this example, you are asking for the status of two print requests with the ID numbers `laser-6885` and `printer_1`.

In addition, by using various options, you can request the following types of information from **lpstat**:

- the status of client printers
- a list of available pre-printed forms
- a list of available character sets and print wheels
- a list of available printers
- the security levels of jobs submitted to the printers.

The rest of this section contains instructions for getting these types of information by issuing the options to **`lpstat`**.

NOTE

> The **`lpstat`** command reports information only on jobs at a security level dominated by your current login level. If you submit a job, log out, and log in at a different level, you may not be able to get a report on the status of your job by running **`lpstat`**.

What Is the Status of the Printers?

First, if you do not already know them, you may want to find out the names of the printers in your system. Which printers are available to you depends on your UNIX system facility. Ask your system administrator for the names of available printers, or type the following command line:

```
$ lpstat -p all <RETURN>
```

A list of printers will be displayed, showing which printers are enabled and which are disabled, as follows:

```
printer printer_1 enabled since Aug 22 16:00. available.
printer printer_1 disabled since Aug 26 22:00. available.
```

If you already have the names of the printers on your system, you can get a status report on one or more of them by listing the appropriate names in place of the argument `all` in the preceding example.

```
$ lpstat -p printer_1,printer_3 <RETURN>
```

More detailed status reports can be obtained by adding the **`-l`** option to the **`lpstat`** command line, as follows:

```
$ lpstat -p printer_1,printer_3 -l <RETURN>
```

For each printer you have specified, a status report will be displayed. Each report will include the following: the printer type, the types of forms allowed and mounted on it, acceptable content types, the names of users allowed to use the printer, the default dimensions for page size and character pitch, and so on.

The system administrator may restrict access to certain printers. If you are not allowed access to a printer for this reason, the phrase `not available` will appear.

Which Forms Are Available?

To find out which pre-printed forms are available on your system, issue the **`lpstat`** command with the **`-f`** option and the argument `all`, as follows:

```
$ lpstat -f all <RETURN>
```

The command prints a list of all the forms your system recognizes and can handle. Forms mounted on printers in your system are identified as follows:

```
form payroll_check is available to you, mounted on
printer4
```

Forms that are recognized and can be handled by your system but that are not mounted on printers are listed as follows:

```
form payroll_check is available to you
```

The system administrator may restrict access to certain forms. If you are not allowed access to a form for this reason, the phrase `is not available to you` will appear.

If you want to know whether specific forms are available on your system, list them after the **`-f`** option in place of the argument `all`, as in this example:

```
$ lpstat -f laser2,laser2 <RETURN>
```

If you want detailed information about any or all of the available forms, use the **`-l`** option with **`lpstat -f`**, as follows:

```
$ lpstat -f all -l <RETURN>
```

A description of each form, including page length, page width, number of pages, ribbon color, and so on, will be displayed.

Which Character Sets or Print Wheels Are Available?

First, you may want to find out which character sets and/or print wheels are available on your LP print service. Issue the **`lpstat`** command with the **`-S`** option and the argument `all,` as follows:

```
$ lpstat -S all <RETURN>
```

A list of all character sets and print wheels that can be used on printers in your system will be displayed. If you want to know whether one or more specific character sets or print wheels are available, list them on the command line in place of the argument `all`.

```
$ lpstat -S "charset_1 wheel_3" <RETURN>
```

The double quotes that appear around the two arguments `(charset_1` and `wheel_3)` to the **`-S`** option are necessary because these arguments are separated by a space. (If the arguments are separated by commas, double quotes are not required.)

To obtain detailed output from the **`lpstat`** command, add the **`-l`** option to the command line. The output will include the following information about each item specified: a list of the printers on which each character set or print-wheel is available, whether the character set or print-wheel is mounted, and what built-in character set it maps.

What Is the Security Level of a Print Job?

If you need to see the security level of a print job, use the **`-z`** and **`-Z`** options of **`lpstat.`** The **`-z`** makes **`lpstat`** print the security level alias name of the level of the print jobs, while **`-Z`** prints the fully qualified level name.

Changing a Print Request

Suppose you have just noticed that when submitting a request to the print service a little while ago, you forgot to request a longer than usual page length for the job, as you had originally planned to do. Don't worry; it may not be too late to change your request! As long as the job has not actually been printed, you may submit changes to your original request. Simply execute the **`lp`** command again, this time including the **`-i`** option, followed by the request-ID assigned to your request. The **`-i`** option signals your intent to change the previous request to the printer.

For example, suppose your original request was for a page length of 50, a width of 70, no banner, and three copies:

```
$ lp -d printer_2 -o “length=50 width=70 nobanner” -n
3 july.report
request id is printer_2-23
```

(The second line in the above example is the response from the system to your command line.) When you later remembered to request a longer page, you reissued the command as follows:

```
$ lp -i printer_2-23 -o “length=60 width=70 nobanner”
<RETURN>
```

Notice that although there were two options in the original command line (**`-o`** and **`-n`**), only one of them (**`-o`**) is included in the change request. A change request should specify only those options from the original command line that you want to change.

However, as this example also shows, when changing the values in a **`-o`** option, you must not only request additional arguments or request different arguments in place of existing ones, you must also repeat those arguments you want to preserve. (This requirement also applies to the **`-y`** option.) Look again at the command lines in the preceding example. Notice that three arguments are given for the **`-o`** option: `length`, `width`, and `nobanner.` Although only one argument to **`-o`** is being changed (from `“length=50”` to `“length=60”`), all three arguments are listed in the change request. Repeating the `width` and `nobanner` arguments is necessary; they are not otherwise preserved from the original command line.

Canceling a Request: The cancel Command

You can cancel a print request that has already been submitted to the print service as long as you are the person who submitted the print request, your current login security level is the same as the level of the print job, and you request the cancellation on the same system or a network server where the print request was submitted. To stop a request, run the **`cancel`** command.

You can execute the **`cancel`** command with either of two types of arguments: request IDs or printer names. To cancel one print request, run **`cancel`** *request_ID*. To cancel only the job that is currently printing, run **`cancel`** *printer_name;* no other requests in the queue for the named printer will be canceled. Arguments of both types may be intermixed.

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

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

If you want to cancel more than one print request, use the **`-u`** option (followed by your login name) after the **`cancel`** command. To cancel all the requests you submitted to a particular printer, type the following:

```
cancel -u login_name printer
```

To cancel all the requests you submitted to all printers, type the following:

```
cancel -u login_name all
```

Once the **`cancel`** command has been run, the specified job is removed from the queue.

You can invoke this command anytime before a print job has been completed.

NOTE

Your current login security level must be the same as the level of the print job for you to cancel it successfully. If you submit a job, log out, and then log in again at a different security level, you will not be able to cancel the job you submitted earlier.

Customizing Printed Output with the lp Command

The LP print service allows you to determine the appearance of your printed output by using any of numerous options to the **`lp`** command. This section describes those parameters for which you can specify values:

- content type
- page size and pitch settings
- whether to have breaks between multiple files

- whether to have a “banner” page printed with your output
- whether to have your text printed on plain paper or on pre-printed forms (such as invoices or invitations)
- a non-standard character set or print wheel
- miscellaneous job specifications (such as one-sided or two-sided printing) known as “special modes”
- number of copies.

Selecting the Content Type

To print a file, a printer must be capable of correctly interpreting the file's contents. Different printers have different capabilities in this sense; not every printer is able to print every type of content. You can make sure the LP print service assigns your request to a printer capable of printing it by using the **`-T`** option to the **`lp`** command.

The **`-T`** option allows you to specify the format of the content of the file to be printed. For example, suppose you want to print a file containing your monthly report for July (**`july.report`**). The contents of this file are arranged in a 455 type format, which means they can be interpreted by an AT&T Model 455 printer. You know your system has several 455 printers but you don't know the names of any of them. The **`-T`** option lets you request a Model 455 printer without specifying one by name, as follows:

```
$ lp -T 455 -d any july.report <RETURN>
```

The **`-T`** option instructs the print service to select any printer that can print a file with contents of type 455. If you want a particular printer to be used—even if it is the default printer—use the **`-d`** option to identify the printer.

What happens if there are no Model 455 printers? The answer depends on whether any filters have been defined for your system. (Your system administrator can tell you whether any filters are available.) A filter is a program that converts data from one format to another; in this case, from the format in which it was typed in the file to a format that can be “read” by a printer. If there are no printers that can handle the content type of your file, and some filters have been defined for your system, your print request will be sent to a filter. (If necessary, your file will be sent to multiple filters. For example, your file could be sent through a `troff` filter, a `postscript` filter, a `page selection` filter, and a filter for downloading fonts, all of which perform different operations on your file.) The contents of the file will be converted, by the filter, to a content type the printer can handle. If, however, there is no printer that can handle the content type of your file, and there is no filter that will convert the file, your print request will be rejected.

Filters make it possible to have files printed by a variety of printers. There may be situations, however, in which the content type is a critical factor of the job. In such a case you do not want to have a file printed unless it can be printed with the original content type. If your system supports filters and you do not want your print request to be sent to one, specify the **`-r`** option after the **`-T`** option to the **`lp`** command, as follows:

```
$ lp -T 455 -r july.report <RETURN>
```

Note that with the **`-r`** option, if your print request cannot be handled by any printer on your system (because of content type), your print request will be rejected.

NOTE

Filters are installed and maintained on your LP print service by your system administrator. Ask your administrator for a list of content types available to your system.

Defining the Page Size and Pitch Settings

Page size consists of two measurements: length and width. Pitch settings are specifications for the number of lines per inch (vertical measurement) and the number of characters per inch (horizontal measurement). When a file is printed, these dimensions may be determined in one of the following four ways:

- by the printer's default dimensions
- by the default dimensions established by your system administrator
- by the dimensions provided with a particular form you have selected
- by your specification for that particular job.

To request your own specification for a print job, use the **`-o`** option to **`lp`**, and specify the desired sizes in "scaled decimal numbers."

The term "scaled-decimal-number" refers to a non-negative number used to show a unit of size. (The type of unit is shown by a "trailing" letter attached to the number.) Three types of scaled decimal numbers are discussed for the LP print service: numbers that show sizes in centimeters (marked with a trailing "c"), numbers that show sizes in inches (marked with a trailing "i"), and numbers that show sizes in units appropriate to use (without a trailing letter), such as lines, columns, lines per inch, or characters per inch.

The following command line shows how to request a print job with your own specifications for page size and pitch settings. (Specifications are shown in *sdn* or scaled decimal numbers.)

```
$ lp -d any -o "length=sdn width=sdn lpi=sdn\
cpi=sdn" filename <RETURN>
```

Your job will be printed according to the default dimensions for the type of printer you are using under either of two circumstances: (1) if you do not specify page dimensions for your print request; or (2) if you do not use a printer for which specific dimensions have been defined by an administrator. These default dimensions are listed in a database called Terminfo; your system administrator is responsible for maintaining this database and can give you details about it.

For example, if you are using an AT&T Model 455 printer, the default dimensions for your printer will be as follows:

```
Page length: 66 lines
Page width: 132 columns
```

```
    Line pitch: 6 lines per inch
    Character pitch: 12 characters per inch
```

If, however, you are using an AT&T Model 470 printer, the default dimensions will be slightly different:

```
    Page length: 66 lines
    Page width: 80 columns
    Line pitch: 6 lines per inch
    Character pitch: 10 characters per inch
```

Removing Breaks between Files

Your print request may consist of more than one file. By default, the LP print service will assume you want each file to be printed separately. If you want the set of files to be printed continuously, without having each file begin on a new page, specify the **-o** option, as follows:

```
    $ lp -d any -o nofilebreak filenames <RETURN>
```

Eliminating the Banner Page

On a system that supports Mandatory Access Control, it is not possible to cancel the banner page. The **-o nobanner** option is ignored.

Controlling Security Labeling of the Output

The banner and trailer pages for each job contain complete information on the security level of the data printed. If the fully qualified level name is too long to fit on a single page, it is printed on two (or more) banner and trailer pages. The LP system also adds a randomly generated number to the banner and trailer pages of each job. This number, which is not the same as the print job ID, is used to ensure that the printer operator can always distinguish the start and the end of each job. It is not possible to remove security level information from the banner and trailer pages or to eliminate the printing of banner and trailer pages.

In addition to the security level information on the banner and trailer pages, the LP system, by default, prints the security level of the data in the header and footer on each page of paginated output. (If the security information does not fit on one line, it is truncated.) If you do not want this information printed on your output, use the **-o** option, as follows:

```
    $ lp -d any -o nolabels filename <RETURN>
```

Use of this option is audited.

Using Pre-printed Forms

Many companies frequently need to issue specialized documents, such as payroll checks and invoices. The LP print service allows you to print your files on pre-printed forms that your administrator loads on your printer. To find out which, if any, special forms are available on your printer, ask your system administrator. If you want to use a particular form and you know it's available, include the **`-f`** option on the **`lp`** command line, followed by the name of the form. For example, say you want to have a file called **`april.payroll`** printed on a type of form called `paycheck` by a printer called "printer4." Enter the following:

```
$ lp -d printer4 -f paycheck april.payroll <RETURN>
```

If the printer you have requested is not capable of handling this form, your request will be rejected. To make sure your request is accepted by any printer on which the desired form can be mounted, include the **`-d`** option, followed by the argument `any`, as shown in the following command line:

```
$ lp -d any -f form_name filename <RETURN>
```

The LP print service will then send your request to any printer capable of handling the type of form required for your job. If your LP print service contains both client and server printers, the command will try to send your job to a client printer before sending it to a server printer.

Using a Character Set or Print Wheel

The **`lp`** command allows you to select a character set or print wheel with which your job will be printed. To find out which character sets and print wheels are available on your system, run the command **`lpstat -S`**.

To request a character set or print wheel for your print job, include the **`-S`** option on the **`lp`** command line, as follows:

```
$ lp -d any -S character_set filename <RETURN>
```

If you have no preference, and if you haven't chosen a form that requires a particular character set or print wheel, you can skip this option.

Special Printing Modes

The final appearance of the document you are printing depends not only on its content, but also on certain other features that affect the composition of the page. For example, you might want to have an unusual font used in your document. The number of special printing modes available to you depends on the available printer(s).

To request special printing modes for your print job, include the **`-y`** option on the command line, as follows:

```
$ lp -d any -y list_of_modes filename <RETURN>
```

Each item in the list of modes must be a one-word name consisting of any combination of letters and numbers.

The printer will accept your request if all the modes you requested in the list are known by the “filter” being used as an interface between your print request and the printer. To find out which filters are available on your system, and which **`-y`** options are allowed, check with your system administrator.

Depending on the print request, one or more filters are used to convert the content type of a file submitted for printing into a type accepted by the printer. Any modes given in the print request are mapped into options understood by the filters. If all modes given in the print request are mapped, the print request is accepted and the LP print service assumes that the filter takes care of invoking the modes on the printer. If one or more modes are not mapped, the print request is rejected.

In choosing qualified printers, the LP print service will ensure that all modes are mapped for the filters needed by each printer, if possible. If more than one printer qualifies on other print request needs, a request is rejected only if one or more modes cannot be mapped on each of those printers.

Each mode is named with a single word, chosen from the set of letters and numbers. No other restrictions are placed on the allowable mode names.

Copies to Be Printed

Some filters allow you to specify a list of pages to have printed, so that you need not print an entire file to obtain a subset of it. Perhaps you want to proofread a section you have edited, give an excerpt of a file to someone, or print the portion of a file that remains unprinted after a print job has been interrupted. With the proper filter, you can limit the printing of a file to a subset of pages by using the **`-P`** option to **`lp`**.

For example, suppose you have a thirty-page business report in a file called **`july.sales`**. Your boss wants to include a copy of the summary and a few of the charts from your report in a package of materials she's putting together for a new director in your division. Because the charts and summary appear on a total of five pages, you don't want to print a copy of the entire thirty-page report. Fortunately, your printer has a filter that allows you to specify a list of pages to be printed. You request only pages 4-6 (for the charts) and 28-29 (for the summary).

```
$ lp -P 4-6,28,29 july.sales <RETURN>
```

If you do not have any filters, or if your filters do not accept a list of pages to print, any requests you make with the **`-P`** option will be rejected and you will be notified of the failure.

NOTE

Your system administrator installs and maintains filters for your system. Check with your administrator to find out if filters are available and whether they will accept the **`-P`** option and lists of pages to be printed.

By specifying a list of pages with **-P**, you can request that printing be started in the middle of a file and that certain pages be skipped. You can present your list of pages in any order; the pages will be printed in order of ascending page number. Also, the LP print system will drop any duplicate requests for pages so that only one copy of each page will be printed.

If you do not include the **-P** option on the command line, the entire file will be printed.

Requesting Multiple Copies

If you want to have more than one copy made, you can request a multiple printing by issuing the **-n** (“number”) option. For example, to have four copies made, enter a command line such as the following:

```
$ lp -d any -n 4 filename <RETURN>
```

When you do not use this option, only one copy is made by default.

Using PostScript Printers

PostScript® is a general purpose programming language, like C or Pascal. In addition to providing the usual features of a language, however, PostScript allows you to specify the appearance of both text and graphics on a page in ways that are more sophisticated than those allowed by other printers. For example, you can create geometric figures and place them anywhere on a page in any size, arranged at any angle. For your text, you can use a variety of fonts in any position, size, or orientation on a page. Graphics and text can be combined easily. In addition, PostScript files can be printed on either low-resolution or high-resolution printers. In short, PostScript printers allow you to produce more varied and sophisticated looking documents than other printers.

PostScript files can be printed only on PostScript printers. These printers are actually special purpose computers capable of interpreting PostScript language files. Unless special provisions have been made by a printer manufacturer, files submitted to a PostScript printer must be written in the PostScript language. However, it is not necessary for you to write files in PostScript.

NOTE

Many popular software packages, including word-processing, spreadsheet, desktop publishing, and computer-aided design packages, support PostScript. If your computer runs one of these packages, you need only create a file in the usual way; the software will translate it into PostScript. To find out whether your software supports PostScript, ask your system administrator.

Once the PostScript printers and filters have been installed, LP manages PostScript files like any others. To request that a PostScript file be printed on a PostScript printer, simply specify the appropriate printer on the command line, and identify the file content type, as follows:

```
$ lp -d ps_printer -T postscript ps_file
```

As long as the printer (*ps_printer*) has been defined with the LP print service as a PostScript printer, the print service will schedule your request and transmit it to the printer.

Support of Non-PostScript Print Requests

The LP print service offers a "translation service": you can create a file in the format you usually use, and the print service will translate it into PostScript language before sending it to a PostScript printer. The print service does this by passing your file through a filter that translates from the "content type" (the formatting language) used in your file to PostScript. Having these filters available means you can use PostScript printers while continuing to write files with your usual formats.

Because each content type requires a separate filter, and UNIX system users create files with many different content types, the print service has many filters for translating files to PostScript. Therefore, if you want to have a file translated, you must request a translation and specify the content type of your file when you submit your print request (that is, when you issue the **`lp`** command). The following is a list of content types that require translation before they can be handled by a PostScript printer.

`troff`	Print the output from `troff.`
`simple`	Print an ASCII ("simple") text file.
`dmd`	Print the contents of a bit-mapped display from a terminal such as an AT&T 630.
`tek4014`	Print files formatted for a Tektronix 4014 device.
`daisy`	Print files intended for a Diablo 630 ("daisy-wheel") printer.
`plot`	Print plot-formatted files.

For example, to convert a file containing ASCII to PostScript code, the filter takes that text and writes a program around it, specifying printing parameters such as fonts and the layout of the text on a page.

The filters that do these translations are invoked automatically by LP when a user specifies one of the content-types listed above for a print request with the **`-T`** option. For example,

```
$ lp -d postprinter -T simple report2
```

automatically converts the ASCII file **`report2`** (a file with an ASCII or "simple" format) to PostScript (as long as the destination printer `postprinter` has been defined to the system as a PostScript printer). The default content-type is `simple`.

Additional PostScript Capabilities Provided by Filters

The filters previously described also take advantage of PostScript capabilities to provide additional printing flexibility. Most of these features may be accessed through the "mode option" (invoked by the **-y** option) to the **lp** command. These filters allow you to use several unusual options for your print job. Table 8-1 describes these options and shows the option you should include on the **lp** command line for each one.

Table 8-1. LP Command Options

Content Type	Type of Print Request
-y reverse	Reverse the order in which pages are printed.
-y landscape	Change the orientation of a logical page from portrait to landscape.
-y x=x*number*,y=y*number*	Change the default position of a logical page on a physical page by moving the origin.
-y group=*number*	Group multiple logical pages on a single physical page.
-y magnify=*number*	Increase or decrease the size of the logical page.
-P *number*	Select, by page numbers, a subset of a document to be printed.
-n *number*	Print multiple copies of a document.

NOTE

If these filters are to be used with an application that creates PostScript output, make sure the application conforms to the PostScript file structuring comments. In particular, the beginning of each PostScript page must be marked by the comment "`%%Page:label` *ordinal*" where *ordinal* is a positive integer that specifies the position of the page in the sequence of pages in the document.

For example, say you have a file called **`report2`** that has a content type `simple` (meaning the content of this file is in ASCII format). You want to print six pages of this file (pages 4-9) with two logical pages on each physical page. Because one of the printers on your system (`postprinter`) is a PostScript printer, you can do this by entering the following command:

```
$ lp -d postprinter -T simple -P 4-9 -y group=2 myfile
```

The filter that groups these logical pages tries to position the pages on the physical page to maximize space utilization. Thus, the pages are printed side by side, so the physical page appears in landscape mode. Landscape mode, which controls the orientation of the logical page rather than the physical page, causes the logical pages to be positioned one on top of the other.

In addition, the LP print service offers a special filter that can print a gray-scale representation of a matrix. (A gray-scale representation of a matrix is a picture in which each cell is colored one of seven shades of gray to show the value of the cell. Darker shades correspond to larger values.) To print a gray-scale representation, specify `matrix` as the content type of your source file by giving the **`-T`** option (**`-T`** matrix).

The dimension of the matrix is assumed to be the square root of the number of elements in the matrix unless you specify the number of rows and columns in it by using the **`-y`** *dimen*=*nrowsxncols* option. The cell values represented by each level of gray may be specified by **`-y`** *interval*=*slash-separated list*. The default list is `-1/0/1`. This separates the elements of the matrix into seven regions: `x<-1`, `x=-1`, `-1<x<0`, `x=0`, `0<x<1`, `x=1`, `x>1`. The list may contain a maximum of three numbers.

How to Use PostScript Fonts

One of the advantages of PostScript is that it allows you to manage fonts. Fonts are stored in outline form, either on a printer or on a computer that communicates with a printer. When a document is printed, the PostScript interpreter generates each character as needed (in the appropriate size) from the outline description of it. If a font required for a document is not stored on the printer being used, it must be transmitted to that printer before the document can be printed. This transmission process is called "downloading fonts."

Fonts are stored and accessed in several ways.

- Fonts may be stored on a printer. The fonts may reside permanently on the printer's disk, or they may be loaded by the administrator into the printer's memory each time the printer is turned on. Ask your print service administrator for a list of fonts available on your printers.
- Fonts may be stored in your own directory, so they're available for your print requests. When you issue a print request that requires a font from your own directory, the font will be transmitted to the printer, along with the source file, as part of your request. This arrangement is preferable for fonts that are not used frequently. Generally, the application program that creates the PostScript file will prepend the font to your PostScript file before delivering it to the print service.
- Fonts may be stored in a public directory on a system shared by many users. These fonts are described as "host-resident." To access these fonts, a user requests fonts to be printed through an application program that creates a PostScript document. When the application program creates a PostScript document file, it must include requests for any desired fonts. This method is useful when the number of fonts is too large to store on the printer.

The LP print service allows you to manage fonts with any of these methods.

The LP print service provides a special download filter to manage fonts using the last method described above. The print service manages this process for you automatically.

Downloading Host-Resident Fonts

The filter that downloads host-resident fonts performs the following tasks:

- It searches the PostScript document to determine which fonts have been requested. Font requests appear in the header comments in the format of PostScript structuring comments:

  `%%DocumentFonts:` *font1 font2* **`. . .`**

- It searches the list of fonts resident on the destination printer to see if the requested font must be downloaded. If the font is not resident on that printer, the filter searches the directory containing host-resident fonts to see if the requested font is available. If it is, the filter takes the file for that font and prepends it to the file to be printed.

NOTE

The download filter relies on the PostScript structuring comments to determine which fonts must be downloaded. If you plan to use this downloading option, make sure the font requests in your application program conform to the PostScript structuring conventions.

Summary of the LP Print Service Commands

Table 8-2l lists the print commands and their functions.

Table 8-2. Print Commands and Their Functions

Command	Function
`lp`	Request paper copies of files from a printer.
`cancel`	Cancel requests for paper copies of files.
`lpstat`	Display information on the screen about the current status of the LP print service.
`enable`	Activate the printers specified so jobs requested through the **`lp`** command can be printed.
`disable`	Deactivate the printers specified so jobs requested through the **`lp`** command can no longer be printed.

Table 8-3 through Table 8-7 summarize the syntax and capabilities of each of these commands.

Table 8-3. Summary of the lp Command

Command Recap **`lp`** - request paper copies of files from a printer		
command	*options*	*arguments*
`lp`	(*as listed*)	*file(s)*
Description:	The **`lp`** command requests that specified files be printed by a printer, thus providing paper copies of the contents.	
Options:	**`-d`** *dest*	Use *dest* as the printer or class of printers to produce the paper copy. You do not have to use this option if the administrator has set a default destination or if you have set the LPDEST environment variable.
	`-f` *form*	Print files on the specified pre-printed form.
	`-H` *special-handling*	Print the request according to the value of *special-handling*. Acceptable values for *special-handling* include: `hold`: Don't print the request until notified. If printing has already begun, stop it. Other print requests will go ahead of a held request until it is resumed. If the Auditing Utilities are installed, the use of this option is an auditable event. `resume`: Resume a held request. If it had been printing when held, it will be the next request printed, unless subsequently bumped by an `immediate` request. If the Auditing Utilities are installed, the use of this option is an auditable event. The **`-i`** option (followed by a *request-ID*) must be used whenever this argument is specified. `immediate`: (Available only to LP administrators) Print the request next. If more than one request is assigned `immediate`, the most recent request will be printed first. If another request is currently printing, it must be put on hold to allow this immediate request to print.
	`-i` *req_ID*	Change specifications for a print request issued but not yet printed.
	`-m`	Send mail when the print job is complete.
	`-n` *copies*	Print the specified number of copies.

Table 8-3. Summary of the lp Command (Cont.)

Command Recap **lp** - request paper copies of files from a printer		
command	*options*	*arguments*
lp	(*as listed*)	*file(s)*
Options:	**-o** *option*	Define page dimensions (length and width, character pitch, and line pitch) as specified. [**-o** performs other tasks, too. See **lp(1)** in the online *Command Reference.]*
	-P *pages*	Print the specified subset of pages. (This option requires a special filter; check with your system administrator to find out whether your system has an appropriate filter.)
	-q *level*	Print the requested job at the specified priority level.
	-S *char_set*	Use the specified character set or print wheel.
	-T *type*	Print the specified content type.
	-w	Display a message on the screen when the print job is complete.
	-y *mode*	Use special printing modes, such as portrait or landscape. (This option requires a special filter; check with your system administrator to find out whether your system has one.)
Remarks:	You can cancel a request to the printer by typing `cancel` and the request ID given to you by the system when the request was acknowledged. Check with your system administrator for information on additional and/or different commands for printers that may be available at your location.	

Table 8-4. Summary of the lpstat Command

Command Recap		
`lpstat` - display information about the status of the **`LP`** print service		
command	*options*	*arguments*
`lpstat`	(*as listed*)	
Description:	The **`lpstat`** command reports the status of print requests, printers, and the LP request scheduler, and provides other information related to the status of the print service.	
Options:	**`-a`** *list*	Report whether print requests are being accepted by specified printers or classes of printers.
	`-c` *list*	Display, for each class in the list, members (printers) of the class.
	`-d`	Show the default destination for the LP print service.
	`-f` *list* [**`-l`**]	Show whether the forms named in *list* are available and where they are mounted. The **`-l`** option lists the form descriptions.
	`-o` *list* [**`-l`**]	Report the status of print requests. *list* may include the names of printers or printer classes, or request IDs.
	`-p` *list* [**`-D`**] [**`-l`**]	Report the status of the printers named in *list*. The **`-D`** option shows a description of each printer, and **`-l`** displays a full description of each printer's configuration.
	`-R`	Show the rank of all print requests in the queue.
	`-r`	Report the status of the LP request scheduler.
	`-s`	Print a status summary of the whole LP print service.
	`-S` *list* [**`-l`**]	Show whether the character sets or print wheels named in *list* are available and, for print wheels, where they are mounted. The **`-l`** option requests a list of printers that can handle each character set and print wheel.
	`-t`	Print all status information.
	`-u` *list*	Report the status of users' print requests. *list* is a list of login names.
	`-v` *list*	List printers and the pathnames of the devices associated with them. *List* is a list of printer names.

Table 8-4. Summary of the lpstat Command (Cont.)

Command Recap **`lpstat`** - display information about the status of the **`LP`** print service		
command	*options*	*arguments*
`lpstat`	(*as listed*)	
Options:	**`-Z`**	Print the fully qualified level name of the security level of each print job. (Valid only if the Enhanced Security Utilities are installed.)
	`-z`	Print the security level alias name of the level of each print job. (Valid only if the Enhanced Security Utilities are installed.)
Remarks:	In each case where *list* is specified, you have the choice of providing a list or specifying all. If you do not specify any options, the -o option is assumed.	

Table 8-5. Summary of the cancel Command

Command Recap **`cancel`** - cancel print requests made by **`lp`**		
command	options	arguments
`cancel`	(as listed)	
Description:	The **`cancel`** command cancels print requests made by the **`lp`** command.	
Options:	[*request-ID* . . .]	Cancel print requests already issued, as specified by the ID numbers of those requests.
	[*printers*]	Cancel whatever print request has been submitted that is currently being printed on the printer specified.
	[**`-u`** *login_name*] [*printers*]	Cancel all jobs submitted by *login_name*. If *printers* are specified, cancel all jobs submitted by *login_name* for the *printers* specified. (Requests submitted to other printers remain unchanged.)

Table 8-6. Summary of the enable Command

Command Recap **`enable`** - activates any printers in an **`LP`** print service		
command	*options*	*arguments*
`enable`	(*as listed*)	*printer(s)*
Description:	The **`enable`** command activates a printer that is part of the LP print service. Your system administrator may or may not authorize users on your system to execute this command.	
Options:	none.	
Remarks:	Run **`lpstat`** to determine the status of printers.	

Table 8-7. Summary of the disable Command

Command Recap **`disable`** - deactivate any printers in the **`LP`** print service		
command	*options*	*arguments*
`disable`	(*as listed*)	*printer(s)*
Description:	The **`disable`** command deactivates a printer that is part of the LP print service. Your system administrator may or may not authorize users on your system to execute this command.	
Options:	**`-c`**	Cancel any requests currently printing on any of the designated printers. (This option cannot be used with the **`-W`** option.)
	`-r` *reason*	Assign a *reason* for the disabling of the printers. This *reason* applies to all printers mentioned up to the next **`-r`** option. This *reason* is reported by **`lpstat`**. If the **`-r`** option is not present, then a default reason will be used.
	`-W`	Wait until the request currently being printed is finished before disabling the specified printer. (This option cannot be used with the **`-c`** option.)

9 Programming with the UNIX System Shell

9
Programming with the UNIX System Shell

Introduction

This chapter shows you how the UNIX system shell can help you do routine tasks. For example, it tells you how to use the shell to manage your files, to manipulate file contents, and to group commands together in programs.

The chapter is organized in two major sections. The first section, *“Shell Command Language,”* describes the use of 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,”* details the use of the shell as a programming language. It tells you how to create, execute, and debug programs made up of commands, variables, and programming constructs such as loops and case statements. Finally, it tells you how to modify your login environment.

To get the most benefit from this tutorial you should log in to your UNIX system and recreate the examples as you read the text. Keep in mind that throughout this book, *italic* and `constant width` are used to distinguish substitutable text from literal input and output. For details see *“Notation Conventions”* in Chapter 1, “*Introduction*” in this book.

Exercises are provided after the *“Shell Command Language”* and *“Shell Programming”* sections. Answers are listed at the end of the chapter.

NOTE

Your UNIX system might not have all the commands referenced in this chapter. If you cannot access a command, check with your system administrator to find out whether it's available.

Normally typical users are logged in under the Korn Shell (ksh) and system administrators under the Bourne shell.

Shell Command Language

This section introduces commands and, more importantly, special characters 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 (of files and other commands)
- terminate running programs.

Table 9-1 summarizes the characters that have special meanings in the shell.

Table 9-1. Characters with Special Meanings in the Shell Language

Character	Function
`* ? [ ]`	The asterisk, question mark, and brackets allow you to specify Filenames by pattern matching.
`&`	The ampersand places commands in background mode, leaving your terminal free for other tasks.
`;`	The semicolon separates multiple commands on one command line.
`\`	The backslash 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 `$` and `` ` ``.
`>`	The greater than sign redirects the output of a command into a file (replacing the existing contents).
`<`	The less than sign redirects the input for a command to come from a file.
`>>`	Two greater than signs redirect the output of a command to be added to the end of an existing file.
`|`	The vertical bar, or pipe, makes the output of one command the input of another command.
``‘...` ``	A pair of grave accents around a command embedded on a command line makes the output of the embedded command an argument on the larger command line.
`$`	The dollar sign retrieves the value of positional parameters and user-defined variables. It's also the default shell prompt.

Filename Generation

The shell recognizes three of the special characters listed in Table 9-1—the asterisk (`*`), the question mark (`?`), and the set of brackets (`[ ]`)—as symbols for patterns that are parts of filenames. By substituting one or more of these characters for the name (or partial name) of an existing file (or group of files), you can reduce the amount of typing you must do to specify filenames on a command line.

The process by which the shell interprets these characters as the full filenames they represent is known as filename expansion. File name expansion is a useful mechanism when you want to specify many files on a single command line. For example, you might want to print a group of files containing records for the month of December, all of which begin with the letters `dec`. By using one of these special characters to represent the parts of the filenames that vary, you can type one print command and specify all the files that begin with `dec`, thus avoiding the need to type the full names of all the desired files on the command line.

This section explains how to use the asterisk, question mark, and brackets for filename expansion.

Matching All Characters with the Asterisk

The asterisk (`*`) matches any string of characters, including a null (empty) string. You can use the `*` to specify a full or partial filename. The `*` alone matches all the file and directory names in the current directory, except those starting with a **.** (dot). 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 filenames in your current directory.

CAUTION

The `*` is a character that matches everything. For example, if you type **`rm *`** you will erase all the files in your current directory. Be very careful how you use the asterisk!

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 display the contents of your report files in order on your screen is by typing the following command:

```
pr report* <RETURN>
```

Now try another exercise. Suppose you have a current directory called **`appraisals`** that contains files called **`Andrew_Adams, Paul_Lang, Jane_Peters`**, and **`Fran_Smith`**, choose a character that all the filenames in your 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 a filename. For example, if you know the first and last letters are the same in several files, you can request a list of them on that basis. If, for example, you had a directory containing files named **`FATE, FE, FADED_LINE, F123E, Fig3.4E, FIRE_LANE, FINE_LINE, FREE_ENTRY,`** and **`FAST_LANE,`** you could use this command to obtain a list of files starting with “`F`” and ending with “`E`.” For such a request, your command line might look like this:

```
ls F*E <RETURN>
```

The system response will be a list of filenames that begin with `F`, end with `E`, and are in the following order:

```
F123E
FADED_LINE
FAST_LANE
FATE
FE
FINE_LINE
FIRE_LANE
Fig3.4E
```

The order is determined by the collating sequences of the language being used, in this case, English: (1) numbers, (2) uppercase letters, (3) lowercase letters.

The `*` is even more powerful; it can help you find all files named **`memo`** in any directory one level below the current directory:

```
ls */memo
```

Matching One Character with the Question Mark

The question mark (`?`) matches any single character of a filename except a leading period (`.`). Let's suppose 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. If your directory contains the following files:

```
chapter1
chapter2
chapter5
chapter9
chapter11
```

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 filenames that match.

Although ? matches any one character, you can use it more than once in a filename. 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 * (asterisk):

```
ls chapter*
```

Matching One of a Set with Brackets

Use brackets ([ ]) when you want the shell to match any one of several possible characters that may appear in one position in the filename. Suppose your directory contains the following files: **cat, fat, mat, rat.** If you include **[crf]** as part of a filename pattern, the shell will look for filenames 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 filenames 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. Suppose you have a directory containing the following files: **chapter1, chapter2, chapter3, chapter4, chapter5** and **chapter6.** If you specify

```
chapter[1-5]
```

the shell will match the 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 displays 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 functions of these special characters are summarized in Table 9-2. Try to use them on the files in your current directory.

Table 9-2. Summary of Filename Generation Characters

Character	Function
*	Match any string of characters (including an empty, or null string) except a leading period.
?	Match any single character, except a leading period.
[*xyz*]	Match one of the characters specified within the brackets.
[*a-z*]	Match one of the range of characters specified.

Special Characters

The shell language has other special characters that perform a variety of useful functions. Some of these additional special characters are discussed in this section; others are described in the next section, *"Input and Output Redirection."*

Running a Command in Background with the Ampersand

Some shell commands take a long 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>

NOTE

You should not run interactive shell commands, such as **read**, 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 associated with the **`grep`** command in the example. You can use this number to terminate the execution of a background command. (Stopping the execution of processes is discussed under *"Executing, Stopping and Restarting Processes.")* The prompt on the last line means that the terminal is free and waiting for your commands; **`grep`** has started running in background mode.

Running a command in background mode 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,"* in this chapter, for details.)

If you want a command to continue running in background after you log out, you can execute it with the **`nohup(1)`** command. (This is discussed under *"Using the nohup Command"* later in this chapter.)

Executing Commands Sequentially with the Semicolon

You can type two or more commands on one line as long as each is separated by a semicolon (`;`) or an ampersand (`&`), 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 pathname of your current directory
3. **`ls`** lists the files in your current directory.

If you want to save the system responses to these commands, (or prevent them from appearing on your screen), you can redirect them to a file. See *"Input and Output Redirection,"* later in this chapter, for instructions.

Turning Off Special Meanings with 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 \ (backslash), the `*` would be expanded by the shell to match all filenames in the current directory.

Turning Off Special Meanings with Quotation Marks

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 except single quotes. Double quotes (`″ . . . ″`) turn off the special meaning of all characters except double quotes, the `$` and the `` ` `` (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 file **`trial`** contained the line

```
trial
```

then

```
grep ????? trial
```

would find the string **`trial`** in the file **`trial`**.

Turning Off the Meaning of a Space with Quotes

Quotes, like backslashes, are commonly used as escape characters 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're 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. **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, together, in double quotes. For example, to print a birthday greeting, 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.

Input and Output Redirection

In the UNIX system, some commands expect to receive their input only from the keyboard (standard input) and most commands display their output at the terminal (standard output). However, the UNIX system lets you redirect both input and output to other files and programs. With such redirection, you can tell the shell to

- take its input from a file rather than from the keyboard
- send its output to a file rather than to the terminal
- use a program as the source of data for another program.

To redirect input and output, you use a set of operators: the less than sign (<), the greater than sign (>), two greater than signs (>>), and the pipe (|).

Redirecting Input with the < Sign

To redirect input, specify a filename after a less than sign (<) on a command line:

command < *file* <RETURN>

When is this mechanism useful? A typical example is when you want to send someone—via the **mail** command—a message or file you've already created. By default, the **mail** command expects input from standard input (that is, the keyboard). But suppose you have already entered the information to be sent (to a user with the login name `jim`) in a file called **report**. Rather than retype that information, you can simply redirect input to **mail** as follows:

```
mail jim < report <RETURN>
```

Redirecting Output with the > Sign

To redirect output (from standard output) to a file, specify a filename after the greater than sign (>) on a command line:

command > *file* <RETURN>

CAUTION

If you redirect output to a file that already exists, the output of your command will overwrite the contents of the existing file.

Before redirecting the output of a command to a particular file, make sure a file by that name does not already exist, unless you do not mind overwriting it. The shell does not allow you to have two files of the same name in one directory. Therefore if you redirect the output of a command to a file with the same name as an existing file, the shell will overwrite the contents of the existing file with the output of your command. Keep this in mind when redirecting output; the shell does not warn you when it is about to overwrite a file.

To make sure that no file exists with the name you plan to use, run the **`ls`** command, specifying your proposed filename 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>
UX:ls:ERROR: Cannot access junk:  No such file or directory
$
```

This means you can name your new output file **`junk`**, but you cannot name it **`temp`** unless you no longer want the contents of the existing **`temp`** file.

Appending Output to an Existing File with the >> Symbol

To keep from destroying an existing file, you can also use the double greater than symbol (>>), as follows:

command >> *file* <RETURN>

This appends the output of a command to the end of the file *file*. If *file* does not exist, it is created when you use the >> symbol this way.

The following example shows how to append the output of the **cat** command, (described in the *"Shell Programming"* section) to an existing file. The **cat** command prints the contents of the files to the standard output. If it has no arguments, it prints its standard input to the standard output. 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>
```

See the **`spell(1)`** manual page for all available options and an explanation of the capabilities of each.

The sort Command

The **`sort`** command arranges the lines of a specified file in alphabetical or numerical order Because users generally want to keep a file that has been alphabetized, output redirection greatly enhances the value of the **`sort`** 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`**.

Combining Background Mode and Output Redirection

Running a command in background does not affect the output of the command; 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, except when an error occurs.

For example, in the section titled *"Special Characters,"* 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 with 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, suppose 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, suppose 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 -c2-13 birthdays <RETURN>
```

The output is shown below:

```
12/26
7/4
10/18
11/9
4/23
8/12
```

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>
Tue Dec 24 13:12:32 EST 1991
$
```

Notice that the time is given between spaces 12 and 19 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
$
```

See the **`date(1)`** manual page for all available options and an explanation of the capabilities of each.

Substituting Output for an Argument

The output of most commands may be captured and used as arguments on a command line. Do this 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 section of this chapter titled *"Shell Programming"* shows you how you can also use the output of a command line as the value of a variable.

Executing, Stopping, and Restarting Processes

This section discusses the following topics:

- how to queue commands to run at a later time with the **batch** and **at** commands
- how to obtain the status of running processes
- how to terminate active processes
- how to restart a stopped process
- how to keep background processes running after you have logged out
- how to move processes in foreground to background, and processes in background to foreground.

Running Commands at a Later Time with the batch and at Commands

The **batch** and **at** commands allow you to specify a command or sequence of commands to be run at a later time. With the **batch** command, the system determines when the commands run; with the **at** command, you determine when the commands run. Both commands expect input from standard input (the terminal); the list of commands entered as input from the terminal must be ended by pressing <CTRL><d> (control-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 <RETURN>
grep dollar * > dol.file <RETURN> <CTRL><d>
UX: At: WARNING: Commands will be executed using /usr/bin/sh
UX; At: INFO: Job 155223141.b at Tue Dec 8 14:24:52 1994
UX: At: WARNING: This job may not be executed at the proper time.
$
```

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.

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 the user with the login name `emily` on her birthday:

```
$ at 8:15am Feb 27 <RETURN>
banner happy birthday | mail emily <RETURN> <CTRL><d>
job 453400603.a at Sat Feb 23 08:15:00 1991
$
```

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 or you can save the job number by redirecting it. The general format is

```
at -r job_number< <RETURN>
```

Try erasing the previous **at** job for the happy birthday banner. Type:

```
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>
168302040.a Mon Nov 25 13:00:00 1991
453400603.a Sun Dec 08 08:15:00 1991
$
```

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 a "here document," described in the section titled *"Shell Programming.")"* Then try the **at** command with the **-l** option.

```
$ at 12:00pm <RETURN>
mail mylogin < memo <RETURN><CTRL><d>
job 263131754.a at Jun 25 12:00:00 1991
$ at -l <RETURN>
263131754.a at Jun 25 12:00:00 1991
$
```

Obtaining the Status of Running Processes

The **ps** command gives you the status of all the processes currently being run. For example, you can use the **ps** command to show the status of all the processes you run in the background mode using & (described earlier in the section titled *"Special Characters")."*

The next section, *"Terminating Active Processes,"* discusses how you can use the PID (process identification) number to stop a command from executing. A PID is a unique number 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  CLS PRI  TTY  TIME  COMD
20723 TS 70  pts000 0:01  sh
20803 TS 59  pts000 0:00  ps
20823 TS 59  pts000 0:01  grep
$
```

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 throughout the time you are logged in. (The shell program **sh** interprets—that is, passes on to the computer—shell commands.)

See the **ps (1)** manual page for all available options and an explanation of the capabilities of each.

You can suspend and restart programs if your login has been configured for job control. See your system administrator to have your login set up to include job control. The **jobs** command also gives you a listing of current background processes, running or stopped.

However, in addition to the PID, the **`jobs`** command gives you a number called the "job identifier" (JID) and the original command typed to initiate the job (*job_name*). You need to know the JID of a process whenever you want to restart a stopped job or resume a background process in foreground. The JID is printed on the screen when you enter a command to start or stop a process. To obtain information about your stopped or background jobs, type:

```
jobs <RETURN>
```

The system will respond by displaying information such as the following:

```
[JID] - Stopped(signal)      job_name
    or
[JID] + Running              job_name
```

Terminating Active Processes

The **`kill`** command terminates active shell processes in background mode and the **`stop`** command temporarily suspends the process if job control is active. The general format for these commands is:

```
kill PID <RETURN>
    or
stop %JID <RETURN>
```

Note that you cannot terminate background processes by pressing the <BREAK> or <DELETE> key. The following example shows how you can terminate the **`grep`** command that you started executing in background mode in the previous example.

```
$ kill 28223 <RETURN>
    [JID] + Terminated job_name
$
```

Notice that 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:

```
UX:sh:ERROR:kill:28223:no such process
```

To suspend a foreground process in the job shell (only when job control is active), type:

```
<CTRL><z>
```

A message appears on the screen resembling the following:

```
[JID]     Stopped(user)      job_name
```

See the **`kill (1)`** manual page for all available options and an explanation of the capabilities of each.

Restarting a Stopped Process

When job control is active you can restart a suspended process. To restart a process with the **`stop`** command, you must first determine the JID by using the **`jobs`** command. You can then use the JID with the following commands:

fg *%JID*	Resume a stopped or background job in foreground.
bg *%JID*	Restart a stopped job in background.

Using the nohup Command

All processes, except the **`at`** and batch requests, are killed when you log out. If you want a background process to continue running after you log out, you must use the **`nohup`** command to submit that background command.

To execute the **nohup** command, use the following format:

nohup *command* & <RETURN>

Notice that you place the **nohup** command before the command you intend to run as a background process.

For example, suppose you want the **`grep`** command to search all the files in your current directory for the string `word` and redirect the output to a file called **`word.list`**, and you want to log out 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. 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 the shell command language. Answers to the exercises appear at the end of the chapter.

Command Language Exercises

1-1. What happens if you use an * (asterisk) at the beginning of a filename? Try to list some of the files in a directory using the * with the last letter of one of your filenames. What happens?

1-2. Try to enter the following two commands:

```
cat [ 0-9 ] * <RETURN>
        echo * <RETURN>
```

1-3. Is it acceptable to use a `?` at the beginning or in the middle of a pattern? 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> trial; ls >> 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 following 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:

```
$ cat testfile <RETURN>
first_command <RETURN>
        .
        .
        .
last_command <RETURN>
$
```

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.

Shell Programs

Creating a Simple Shell Program

We'll begin by creating a simple shell program that will do the following tasks, in order:

- 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 editor of choice, 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-------1login login 3661Nov  210:28 mbox
-rwx------1login login 48  Nov 1510:50 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.

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 reminds 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?
```

It is possible to give the **bin** directory another name; if you do so, you must change your shell variable PATH again.

Warnings about Naming Shell Programs

You can give your shell program any appropriate filename; however, you should not give your program the same name as a system command. Depending on your path, the system may 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 might 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. For this reason, the UNIX system limits the number of times an infinite loop can execute. One way to prevent such looping is to give the pathname for the system **`ls`** command, **`/usr/bin/ls`**, when you write your own shell program.

The following **`ls`** shell program would work:

```
$ cat ls <RETURN>
pwd
/usr/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, **`/usr/bin/ls`**.

Variables

Variables are the basic data objects that, in addition to files, shell programs manipulate. Here we discuss three types of variables and how to use them:

- positional parameters
- special parameters
- named variables.

Positional Parameters

A positional parameter is a variable within a shell program; its value is set from an argument specified on the command line that invokes 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 with 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 is assigned the value `pp1`, positional parameter `$2` within the program is assigned the value `pp2`, and so on, at the time the shell program is invoked.

To practice positional parameter substitution, create a file called `pp` (short for positional parameters). (Remember, the directory in which these example files reside must be in `$PATH`.) 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 (but 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
$
```

Another example of a shell program is **`bbday`**, which mails a greeting to the login entered in the command line. The **`bbday`** program contains one line:

```
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>
```

The **`who`** command lists all users currently logged in on the system. How can you make a simple shell program called **`whoson`**, that will tell you if the owner of a particular login is currently working on the system?

Type the following command line into a file called **`whoson`**:

```
who | grep $1 <RETURN>
```

The **`who`** command lists all current system users, and **`grep`** searches that output for a line with 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, suppose 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 appears on your screen as follows:

```
$ 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.

The shell allows a command line to contain at least 128 arguments; however, a shell program is restricted to referencing only nine positional parameters, `$1` through `$9`, at a given time. You can work around this restriction by using the **`shift`** command. See **`sh(1)`** for details. The special parameter `$*` (described in the next section) can also be used to access the values of all command line arguments.

Special Parameters

`$#` This parameter, when referenced in a shell program, contains the number of arguments with which the shell program was invoked. Its value can be used anywhere in the shell program.

Enter the command line, shown in the following screen, in the 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
$
```

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: $*
$
```

The program **show.param** will echo all the arguments you give the command. Make **show.param** executable and try it 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 a b c d e f g h i j <RETURN>
The parameters for this command are: a b c d e f g h i j
$
```

Once again, **show.param** echoes all the arguments you give. The `$*` parameter can be useful if you use filename expansion to specify arguments to the shell command.

Use the filename expansion feature with your **show.param** command. For example, suppose you have three files in your directory named for the first three chapters of a book. The **show.param** command prints a list of all those files.

```
$ show.param chap? <RETURN>
The parameters for this command are: chap1 chap2 chap3
$
```

Named Variables

Another form of variable that you can use in 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 equals (`=`) 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 consist of letters, underscores, and digits. Like shell program filenames, variable names should not be shell command names. Also, the shell reserves some variable names that you should not use for your variables. The following list provides brief descriptions of some of the most important of these reserved shell variable names.

- `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 `<RETURN>`)
- `OFS` defines the output field separators.
- `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 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

You can set the `TERM` variable by entering the following command line:

```
TERM=terminal_name <RETURN>
export TERM
```

This is the simplest way to assign a value to a variable. However, 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 a `<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	Prompt you for a person's last name.
read	Assign the input value to the variable `name`.
grep	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 home/list
$
```

Create a file called **list** that contains several last names and telephone 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 telephone number you added.

Running the **cat** command on **mknum** displays the contents of the program. 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 telephone list. In the next example, **mknum** creates a new listing for Mr. Niceguy. Then **num.please** gives you Mr. Niceguy's telephone number:

```
$ mknum <RETURN>
Type in name
Mr. Niceguy <RETURN>
Type in 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.

Substituting Command Output for the Value of a Variable

You can substitute the output of a command for the value of a variable by using command substitution in 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 gives you the time.

```
$ cat t <RETURN>
time=`date | cut -c12-16`
echo The time is: $time
$
```

Remember, there are no spaces on either side of the equal sign. Make the file executable, and you'll have a program that gives you the time:

```
$ chmod u+x t <RETURN>
$ t <RETURN>
The time is: 10:36
$
```

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. By running the **`cat`** command on **`simp.p,`** you can see the contents of this program:

```
$ cat simp.p <RETURN>
var1=$1
echo $var1
$
```

Of course, you can also assign to a variable the output of a command that uses positional parameters, 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 $1`
echo $person >> $home/login.file
echo $person
$
```

If you execute **log.time** specifying maryann as the argument, the system responds as follows:

```
$ log.time maryann <RETURN>
maryann     tty61         Apr 11 10:26
$
```

Shell Programming Constructs

The shell programming language has several constructs that give added flexibility to your programs:

- Comments let you document the function of a program.
- The “here document” allows you to include, within the shell program itself, lines to be redirected as 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

When you place comments in a shell program, the shell ignores all text on a line following a word that begins with a # (pound) sign. If the # sign appears at the beginning of a line, the comment uses the entire line; if it appears after a command, 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.
# This program needs a login as
# the positional parameter.
```

Comments are useful for documenting the function of a program 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 to a command in that program. By using a here document, you can provide input to a command in a shell program without using 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.

Screen 9-1 shows the general format for a here document.

command <<*delimiter* <RETURN>
`. . .` *input lines* `. . .` <RETURN>
delimiter <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 $1 <<!
Best wishes to you on your birthday.
!
$
```

Screen 9-1. Format of a Here Document

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>

User `mary` will receive your greeting the next time she reads her mail messages:

```
$ mail <RETURN>
From mylogin Mon May 14 14:31 CDT 1991
Best wishes to you on your birthday.
$
```

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 filename.
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.

The stream editor **sed** can also be used in shell programming.

Return Codes

Most shell commands issue return codes that show whether the command executed properly. By convention, if the value returned is `0` (zero), then the command executed properly; any other value shows 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
UX:cat:Error:Cannot open hello: No such file or directory
$ 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.

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 format shown in the following listing.

```
for variable <RETURN>
        in a_list_of_values <RETURN>
do <RETURN>
        command_1 <RETURN>
        command_2 <RETURN>
            .
            .
            .
        last_command <RETURN>
done <RETURN>
```

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. Because the shell ignores spaces at the beginning of lines, each section of a command 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`	Prompt the user for a pathname to the new directory.
`read`	Assign the pathname to the variable `path`.
`for` *variable*	Call the variable `file`; it can be referenced as `$file` in the command sequence.
`in` *list_of_values*	Supply a list of values. If the `in` clause is omitted, the list of values is assumed to be `$*` (all the arguments entered on the command line).
`do` *command_sequence*	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 filenames as arguments to the command. (This can be done most easily using the filename expansion mechanism described earlier).

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 statements after the `do` can be executed.

The general format of the `while` loop is shown in the following listing.

```
while <RETURN>
        command_1 <RETURN>
            .
            .
            .
        last_command <RETURN>
do <RETURN>
        command_1 <RETURN>
            .
            .
            .
        last_command <RETURN>
done <RETURN>
```

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 **`/dev/null`** by ignoring 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 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` . . . then construct is shown in the following listing.

```
if <RETURN>
        command_1 <RETURN>
            .
            .
            .
        last_command <RETURN>
   then <RETURN>
            command_1 <RETURN>
                .
        .
        .
    last_command <RETURN>
fi <RETURN>
```

For example, a shell program called `search` demonstrates the use of the `if . . . then` construct. The `search` program uses the **`grep`** command to search for a word in a file. If **`grep`** is successful, the program **`echo`**s 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 filename.
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 to 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 general format shown in the following listing.

```
if <RETURN>
    command_1 <RETURN>
        .
        .
        .
    last_command <RETURN>
  then <RETURN>
        command_1 <RETURN>
```

```
                .
                .
                .
            last_command <RETURN>
    else <RETURN>
            command_1 <RETURN>
                .
                .
                .
            last_command <RETURN>
fi <RETURN>
```

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 filename.
read word file
if
   grep $word $file >/dev/null
then
   echo $word is in $file
else
   echo $word is NOT in $file
fi
$
```

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 listed in Table 9-3:

Table 9-3. Test Command Options

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

Table 9-3. Test Command Options (Cont.)

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 is 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 `*` special character as the command argument. The command lines shown in the following example execute 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
$
```

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 is shown in the following listing:

```
case word <RETURN>
in <RETURN>
  pattern_1) <RETURN>
        command_line_1 <RETURN>
                .
                .
                .
        last_command_line <RETURN>
  ;; <RETURN>
  pattern_2) <RETURN>
        command_line_1 <RETURN>
                .
                .
                .
        last_command_line <RETURN>
;; <RETURN>
pattern_3) <RETURN>
        command_line_1 <RETURN>
                .
                .
                .
        last_command_line <RETURN>
 ;; <RETURN>
*) <RETURN>
        command_1 <RETURN>
                .
                .
                .
        last_command <RETURN>
  ;; <RETURN>
esac <RETURN>
```

The `case` construction tries to match the *word* following the word `case` with the *pattern* in the first pattern section. If a match exists, 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 helps you detect incorrect or unexpected input.

The patterns that can be specified in the *pattern* part of each section may use the special characters `*,?,` and `[]` for filename expansion, as described earlier in this chapter. 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>

In the following example, assume the terminal is a Teletype 4420, Teletype 5410, or Teletype 5420.

The **`set.term`** program 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 the value of `term` is not 4420, the program checks for other possible values: 5410 and 5420. It executes the commands under the first pattern it finds, and then goes to the first command after the **esac** command.

The pattern `*`, meaning everything else, is included at the end of the terminal patterns. It warns that you do not have a pattern for the terminal specified and it allows 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 in the preceding screen. 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.

Unconditional Control Statements: the break and continue Commands

The **break** command unconditionally stops the execution of any loop in which it is encountered, and goes to the next command after the `done, fi,` or `esac` statement. If no commands follow 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 `while` or `for` loop without executing the remaining commands in the loop.

Functions

Functions provide a convenient and efficient means of coding and executing simple programs (more complex programs should remain as shell programs). Functions are similar to shell programs except for the fact that they are stored in memory and therefore, execute faster than a program. Another exception is that functions only operate in the current shell process.

Defining a Function

There are two formats that can be used in defining a function:

Format 1

```
name () { command; command; ... command; }
```

In this format, `name` is the name of the function. The parentheses is an indication to the shell that a function is being defined. The body of the function (which is delimited by the curly braces) contains the commands to be executed. Each command is separated by a semi-colon and a space. The last command ends with a semi-colon, and the curly braces are separated from the body of the function by a space.

Format 2

```
name ()
> {
> command
> command
> command
> }
```

In this format, `name ()` is the same as in *format 1*. However, upon pressing the `<RETURN>` key, a `>` prompt will replace your regular shell prompt. The body of the function is coded at this point, starting with the left curly brace. After the last command has been entered, the body of the function is closed with a right curly brace. It is not necessary to use semi-colons in this format.

Just as the `exit` statement is used within shell programs, the `return` statement is provided for use within functions. This statement will terminate the function, but not the shell program that called the function. The format of the return statement is:

`return` *n*

where *n* is the return status of the function. If *n* is omitted, or if a `return` statement is not coded within the function, then the return status is that of the last command executed within the function.

Once the function has been defined, you can display it by using the shell **`set`** statement (without arguments) which displays all of your current environment variable settings. At the end of the variable list, any functions you have defined will be displayed.

If you find it necessary to remove a function during a session, the **`unset`** command can be used.

The format is:

`unset` *function*

where *function* is the name of the function to be removed.

Executing a Function

To execute a function, enter the name of the function at your regular shell prompt. Any arguments listed after the name of the function replace the positional parameters coded within the function (the same as in any other shell program).

After a function has executed, you can display the return status by issuing the following:

```
echo $?
```

Examples

The following defines a function that displays login information for a particular user (notice that *format 1* is used in this case):

```
whoon () { who | grep $1; }
```

The next example searches for a file in the current directory. Notice that *format 2* is used in this case. Also, the return statement is used. A return status of 1 indicates that the search did not find the file in question (a message is also displayed to that effect). A return status of 0 indicates that the file exists.

```
isthere ()
> {
>   if [ ! -f $1 ]
>   then
>       echo “$1 was not created”
>       return 1
>   fi
>   return 0
> }
```

Debugging Programs

At times you may need to debug a program to find and correct errors. Two options to the **sh** command can help you debug a program:

sh -v *shell_program_name*	Print the shell input lines as they are read by the system.
sh -x *shell_program_name*	Print commands and their arguments as they are executed.

To try 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 at login `tommy` (`$1`) should look like the following screen:

```
$ mail <RETURN>
From mlh  Wed Apr 10  11:36  CST  1991
Tom
When you log off come into my office please.
Wed  Apr 10  11:36:32  CST  1991
MLH
?
.
```

To execute **bug**, you 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 tommy <RETURN>
today=`date`
echo enter person
enter person
read person
tom
mail $1
```

Notice 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 tommy <RETURN>
+date
today=Wed  Apr 10  11:07:23  CST  1991
+ echo enter person
enter person
+ read person
tom
+ mail tommy
$
```

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 helpful 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:

command_1 `|` `tee` *saverfile* `|` *command_2* `<RETURN>`

`saverfile` is the file in which the output of *command_1* is saved for you to study.

For example, suppose 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:

```
$ 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. For example, users frequently want to change the default values of the erase and line kill characters, `<CTRL><h>` and `@`, respectively.

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 shell script, it can be edited and changed to suit your needs. On some systems you can edit this file yourself, whereas on others, the system administrator must do this for you. To see whether you have a **.profile** in your home directory, type:

```
    ls -al $HOME <RETURN>
```

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. You can also set some terminal options with the **stty** command, and set some shell variables.

Adding Commands to Your .profile

Practice adding commands to your **.profile.** Edit the file and add the following **echo** command to the last line of the file:

```
    echo Good Morning! I am ready to work for you.
```

Write and quit the editor.

Whenever you make changes to your **.profile** and you want to initiate them in the current work session, you may cause the commands in **.profile** to be executed directly, using the **.** (dot) shell command. The shell reinitializes your environment by 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. You can use these options with **stty**: **-tabs** 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)** for details.)
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 three command lines to your **.profile:**

```
$ tail -3 .profile <RETURN>
echo Good Morning! I am ready to work for you
stty -tabs
stty echoe
$
```

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* <RETURN>

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.

LANG For many commands, this variable gives the language (such as French, German, and so on) in which messages from the system are displayed on your screen. It also specifies the language and cultural conventions the commands will use to process and sort characters, display the date and time, and interpret numeric and monetary values. The default language is English. If you prefer to work in another language, and if your system supports non-English usage, you can specify the desired language with this variable by assigning an appropriate value to it. For example, for German usage, you might enter

```
LANG=de1[utsche]
```

Ask your system administrator which languages are available on your computer, and what values you must assign to LANG to access them. Not all system commands support non-English usage. Check **intro(1)** for the ones that do. For details of LANG usage, see **environ(5).**

`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
        $
```

The colon (`:`) is a delimiter between pathnames in the string assigned to the `$PATH` variable. When nothing is specified before a `:`, the current directory is understood. Notice how, in the last example, the system looks for commands in the current directory first, then in **/mylogin/bin**, then in **/bin**, and finally in **/usr/bin**.

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="$PATH:/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.

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 `$` 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 gives 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 sends 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 as the previous exercise, without erasing the date.

2-6. Using the above exercises, write a shell program that sends a memo to the same people on your system mentioned in Exercise 2-3. Include in your memo:

- lines at the top that include the current date and the words “Dear colleague”
- the body of the memo (stored in an existing file)
- a 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 filename refers to all files that end in that filename, including that filename.

```
$ ls *t <RETURN>
     cat
     123t
     new.t
     t
$
```

1-2. The command **cat** `[0-9]*` produces the following output:

```
     1memo
     100data
     9
     05name
```

The command **echo** `*` produces a list of all the files in the current directory.

1-3. You can place `?` in any position in a filename.

1-4. The command **ls** `[0-9]*` lists only those files that start with a number.

The command **ls** `[a-m]*` lists only those files that start with the letters "a" through "m."

1-5. If you placed the sequential command line in the background mode, the immediate system response was the PID number for the job.

No, the & (ampersand) must be placed at the end of the command line.

1-6. The command line would be:

```
cd; pwd > junk; ls >> junk <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>
```

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 filenames, 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 the hint provided 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. Enter the following commands to check the values of the `HOME`, `TERM`, and `PATH` variables in your home environment:

- $ echo $HOME <RETURN>
- $ echo $TERM <RETURN>
- $ echo $PATH <RETURN>

Summary of Shell Command Language

This appendix is a summary of the shell command language and programming constructs. The first section reviews metacharacters, special characters, input and output redirection, variables, and processes. These are arranged by topic in the order that they were discussed. 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 filenames, through pattern matching.
&	Executes commands in the background mode.
`;`	Sequentially executes several commands typed on one line, each pair separated by `;`.
\	Turns off the meaning of the immediately following special character.
`´...´`	Enclosing single quotes turn off the special meaning of all characters except single quotes.
“...”	Enclosing double quotes turn off the special meaning of all characters except $, single quotes, and double quotes.

Redirecting Input and Output

<	Redirects the contents of a file into a command.
>	Redirects the output of a command into a new file, or replaces the contents of an existing file with the output.
>>	Redirects the output of a command so that it is appended to the end of a file.
|	Directs the output of one command so that it becomes the input of the next command.
`` `*command*` ``	Substitutes the output of the enclosed command in place of `` `*command*` ``.

Executing and Terminating Processes

`batch`	Submits the following commands to be processed at a time when the system load is at an acceptable level. `<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 out.

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 automatically assigned by the shell from the arguments of the command line invoking the shell 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.
`MAIL`	Gives the name of the file containing your electronic mail.
`PS1, PS2`	Defines the primary and secondary prompt strings, respectively.
`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

The format of the Here document is shown in the following listing:

```
command <<!
input lines
!
```

For Loop

The format of the for loop is shown in the following listing:

```
for variable
        in this list of values
do the following commands
        command 1
        command 2
            .
```

```
              .
        last command
done
```

While Loop

The format of the while loop is shown in the following listing:

```
while   command list
do
        command 1
        command 2
              .
              .
        last command
done
```

If...Then

The format of the if. . . then command is shown in the following listing:

```
if this command list is successful
then    command 1
        command 2
              .
              .
        last command
fi
```

If...Then...Else

The format of the if . . . then . . . else format is shown in the following listing:

```
if      command list
        then   command list
        else   command list
fi
```

Case Construction

The format of the case construction is shown in the following listing:

```
case word
in
        pattern1)
                command line 1
                        .
                        .
                last command line
        ;;
        pattern2)
                command line 1
                        .
                        .
                last command line
        ;;
        pattern3)
                command line 1
                        .
                        .
        last command line
        ;;
esac
```

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.

10
Electronic Mail Tutorial

10
Electronic Mail Tutorial

Introduction

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 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: **`mail`**, **`mailx`**, **`uname`**, and **`uuname`** commands to exchange messages.

The chapter will also show you how to use the **`mailcheck`** command to check if you have mail at various levels, and the **`vacation`**, **`notify`** and **`mailproc`** for managing your incoming messages.

To help you exchange files, and for further information on networking, see Chapter 12, the *Communication Tutorial* in this guide.

Exchanging Messages

To send messages you can use either the **`mail`** or **`mailx`** command. These commands deliver your message to a file belonging to the recipient. When the recipient logs in (or while already logged in), he or she receives a message that says `you have mail`. The recipient can use either the **`mail`** or **`mailx`** command to read your message and reply at his or her leisure.

The main difference between **`mail`** and **`mailx`** is that only **`mailx`** offers the following features:

- a choice of text editors (**`ed`** or **`vi`**) for handling incoming and outgoing messages
- a better understanding of mail header fields
- the ability to tailor the mail environment
- the ability to create enhanced text, multimedia, and multi-part messages using the Multi-purpose Internet Mime Extensions (MIME) standard.

You can also use **`mail`** or **`mailx`** to send files containing memos, reports, and so on. However, if you want to send someone a file that is over a page long, it is probably better to use one of the commands designed for transferring files: **`uuto`** or **`uucp`**. (See *“Sending Files to the Public Directory: The uuto Command”* in Chapter 12, the “*Communication Tutorial*” for descriptions of these commands.) These commands are also preferred for sending files which contain non-text characters to a system running an earlier release of the UNIX system.

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 one 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 system that can communicate with yours (for example, `sys2!bob` or `bob@sys2`)
- a system-wide alias name which has been established by your system administrator.

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, type the recipient's login id, press the `<RETURN>` key, 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 (`.`) and `<RETURN>` 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, suppose you (owner of the login `kol`) want to send a message to a user with the login `chris`. 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>
. <RETURN>
UX:mail:ERROR:Can't send to cris
UX:mail:ERROR:Return to kol
you have mail in /var/mail/kol
$
```

The message `you have mailed` is presented by the shell; different shells may use slightly different wordings for this message.

The mail that is waiting for you in your mailbox, which may be found in **`/var/mail/`***your-user-id*, 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 postmaster Mon Jan 23 16:00 EST 1993
To: kol@xquartet
Date: Mon Jan 23 21:00:01 GMT 1993
Original-Date:  Mon Jan 23 15:59 EST 1993
Not-Delivered-To: !recipients due to 02 Invalid/Ambiguous Originator/
Recipient Name
     ORIGINAL MESSAGE ATTACHED
     (mail: Error # 8 'Invalid recipient')
En-Route-To: cris
Content-Length: 177

From kol@xquartet Mon Jan 23 16:00 EST 1993
Date: Mon, 23 Jan 93 16:00 EST
To: cris
Content-Length: 38
Content-Type: text/plain
Message-ID: <2c976ef10.3af7@xquartet>

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 **`/var/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 1992 figures! <RETURN>
. <RETURN>
$
```

The mail will be delivered immediately and remain there until you delete it. 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 several people by including all their login names, separated by white space, 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>
$
```

Sending Mail to Remote Systems: The uname and uuname Commands

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.

If your system has the Basic Networking Utilities or SMTP packages installed, 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 mark.

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 the system names, but you will have to get the recipient's login name from the recipient. If you can, also get the name of the remote system 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, a prompt is returned.

```
$ 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 `jumbo`. Verify that `jumbo` can communicate with your system and send your message. The following screen shows both steps.

```
$ uuname | grep jumbo  <RETURN>
jumbo
$ mail jumbo!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>
$
```

Table 10-1 and Table 10-2 summarize the syntax and capabilities of the **uname** and **uuname** commands, respectively.

Table 10-1. Summary of the uname Command

Command Recap **uname** - displays the system name		
command	*options*	*arguments*
uname	**-n** and others[1]	none
Description:	**uname -n** displays the name of the system you are logged into.	

1. See **uname(1)** in the online *Command Reference* for all available options and an explanation of their capabilities.

Table 10-2. Summary of the uuname Command

Command Recap **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.	

Looking up SMTP Names

In addition to the Basic Networking Utilities package, your system may have the SMTP package installed. To determine if mail would be able to deliver mail to the remote system using SMTP, use grep from the file **/etc/inet/hosts**:

```
$ grep dumbo /etc/inet/hosts <RETURN>
123.45.67.89    dumbo
$
```

If this does not list the system, SMTP may still be able to deliver the mail by using the network itself to look up the system. Use the **`nslookup`** command to determine this. In the following examples, the network is asked if the system `dumbo` exists anywhere. If so, you will be shown an address for that system.

```
$ nslookup dumbo <RETURN>
   . . .
Non-authoritative answer:
Name:    dumbo
Address: 123.45.67.89
   . . .
$
```

In this example, the network is asked to use any Message Exchangers (MX's) that may be available. The greater than sign (`>`) is **`nslookup`**'s prompt.

```
$ nslookup <RETURN>
> set q=mx <RETURN>
> dumbo <RETURN>
   . . .
Non-authoritative answer:
dumbo     preference = 0, mail exchanger = dumbo
dumbo     preference = 5, mail exchanger = jumbo.you.com
   . . .
```

Domain-Style Addresses

In addition to the addressing style described above, using *remote_system!recipient*, another addressing syntax known as domain-style addressing is supported. Here the address would be in the form

recipient@*remote_system*

or

recipient@*remote_system.domain_info*

The above two addresses are equivalent to the addresses

remote_system`!`*recipient*

or

remote_system.domain_info!*recipient*

NOTE

Your local system administrator can set up other addressing schemes which make it unnecessary to verify direct communication with a remote system. This means that if the remote system cannot be contacted directly, your message will be automatically forwarded through a system that can contact the indicated remote system. Check with your local administrator.

Other addressing syntaxes may be set up by your local System Administrator. Your local System Administrator may also have set it up (check with your local System Administrator to be sure), such that it may not be necessary to verify that your local system can directly communicate with the remote system. If the remote system cannot be contacted directly, the message may be automatically forwarded to another system that can service the indicated remote system.

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

Table 10-3. Summary of Sending Messages with the mail Command

Command Recap **`mail`** - sends a message to another user's login		
command	*options*[1]	*arguments*
`mail`	none required	*login*
`mail`	none required	*system_name*!*login*
`mail`	none required	*login@system_name*
Description:	Typing **`mail`** followed by one or more login names (which may include a system name), sends the message typed on the lines following the command line to the specified login(s).	
Remarks:	Typing a period (**.**) (followed by the `<RETURN>` key) or a `<CTRL><d>` at the beginning of a new line sends the message.	

1. See the **`mail(1)`** manual page in the online *Command Reference* for all available options and an explanation of their capabilities.

Managing Incoming Mail

As stated earlier, the **mail** command also allows you to display on your screen messages sent to you by other users 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 **/var/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 <RETURN>
From tommy Wed May 21 15:33 CST 1993
Content-Length: 104

Bob,
Looks like the meeting has been canceled.
Do you still want the material for the technical review?
Tom

?
```

The first set of lines, called the message header, provides information about the message: the login name of the sender, the date and time the message was sent, and how many characters are in the message. The lines after the first blank line (up to the line containing the ?) comprise the contents 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 pause 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 ? at the? prompt and then press the <RETURN> key.

```
? ? <RETURN>
```

To display the next message without deleting the current message, press the <RETURN> key 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, the shell 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, it is then displayed.

To save a message for later reference, type an `s` after the question mark:

```
? s <RETURN>
```

This saves the message 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. The message is also deleted from your mailbox.

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 a directory.)

You can also save the message in a file in a different directory by specifying a path name. For example:

```
? s project1/memo <RETURN>
```

This is a relative path name that identifies a file called **`memo`** (where your message will be saved) in a subdirectory (`project1`) of your current directory. You can use either relative or full path names when saving mail messages. (For instructions on using path names, see Chapter 4,"*Using the File System*".)

To quit reading messages, enter a `q` followed by a `<RETURN>`:

```
? 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 the `<BREAK>` key. The **`mail`** command will stop the display, print a `?` prompt, and wait for a response from you.

Table 10-4 summarizes the syntax and capabilities of the **`mail`** command for reading messages.

Forwarding Mail

It is possible for you to have all of your mail messages forwarded from your login ID to another login ID. This is done by using the **`-F`** option to **`mail`**. The following command says that your mail needs to be forwarded to the two users `bob` and `carol`:

```
mail -F "bob carol"
```

Table 10-4. Summary of Reading Messages with the mail Command

Command Recap **mai**l - read messages sent to your login		
command	*options*	*arguments*
mail	many[1]	none
Description:	When issued without options, the **mail** command displays any messages waiting in your mailbox (the file **/var/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 online *Command Reference.*	

1. See the **mail(1)** manual page in the online *Command Reference* for all available options and an explanation of their capabilities.

To remove the forwarding, use the **-F** option with an empty argument:

```
mail -F ""
```

The vacation, notify, and mailproc Commands

Three other programs related to managing incoming messages are **notify(1), vacation(1)**, and **mailproc(1)**. The **notify** command provides a mechanism for receiving immediate notification of messages as they arrive. To establish notification, issue the command

```
notify -y
```

To turn off notification, issue the command

```
notify -n
```

The **vacation** command provides a way to automatically answer incoming messages with a response while also saving the incoming messages for later perusal. To turn on the **vacation** processing, issue the command

```
vacation
```

Various options are available which allow you to specify the message which will be sent back, and whether the messages will be stored in the normal mailbox or under your home directory. The mail messages can even be split apart into separate sets of messages for each day.

To turn off vacation processing, issue the command

```
vacation -n
```

See the online *Command Reference* for additional details.

The **`notify`** and **`vacation`** commands both use a feature of the mail system known as personal mail surrogates. Using a personal mail surrogate is similar to forwarding your message to another user, except that of the mail going to another person, the mail is instead sent to a program. The **`vacation`** and **`notify`** programs use personal mail surrogates to perform their processing: when you turn on notification or **`vacation`** processing, a personal mail surrogate does the work.

Both **`vacation`** and notification processing use a single command to perform their work, but both these commands cannot be active at the same time.

The **`mailproc`** command is useful because it lets you process your mail using multiple commands. For example, you can write all messages from your boss to a separate file after sending a message to your terminal, send messages from the trouble reporting system directly to the printer, forward messages that contain the word "help" to the user `joe`, and write the rest to your mailbox.

To use the **`mailproc`** command, you must first create a **`$HOME/.mailprocrc`** file, containing lines which indicate how to process each message delivered to your mailbox. Each line of the **`$HOME/.mailprocrc`** file contains four fields separated by colons (`:`):

user`:`*test*`:`*disposition*`:`*action*

user	is a regular expression pattern which is matched against the return address of the mail message; this lets you select mail messages from a particular user.
test	is an arbitrary test which can be used to select mail messages based on a regular expression match against header field values or by running UNIX commands; this lets you select mail messages based on any criteria you choose.
action	specifies what to do when a match is successful in the first two fields; this may be writing the message to a file, or running a UNIX command.
disposition	field specifies what to do with the message after the action is completed successfully or unsuccessfully; it lets you say that the message should be considered successfully delivered and nothing more is to be done with it; that the message was successfully handled, but should be kept in the mailbox; or that the message should be processed by further commands.

For the example above, to capture a message from your boss, use a line similar to the following:

```
boss:::%M>>boss.mail
```

This uses an empty test, which is always true, and the default disposition, which is to consider the mail to be successfully delivered.

To send a message to your terminal telling you that the mail arrived, the following command line would be placed before the previous line (it should be run first):

```
boss::C=*;:echo Your boss sent you mail! | write your-id
```

The disposition field `C=*`; indicates that all exit codes from the UNIX command should be treated such that processing will be continued.

To send all messages from the trouble reporting system, which all have the login id *troubles*, directly to the printer, use a line like this:

```
troubles::S=0;K=*;:lp
```

All mail from the `troubles` login will be sent to the **`lp`** command. The `S=0;` says that if the **`lp`** command succeeds, the message has been successfully delivered. The `K=*;` says that if the **`lp`** command failed for any reason, the message is to be kept in your mailbox.

To forward all other mail messages that contain the word "help" in the subject, you would use the test field with a regular expression match against the subject. Regular expression matches are given surrounded with tildes (~), and the subject text can be extracted using the escape sequence `%s`.

```
:~help~%s~::rmail joe
```

The test `~help~%s~` does the match of the subject against the pattern *help*, and the action is to invoke the **`rmail`** command again to deliver the message to `joe`.

The complete sample **`$HOME/.mailprocrc`** file would look like this:

```
boss::C=*;:echo Your boss sent you mail! | write your-id
boss:::%M>>boss.mail
troubles::S=0;K=*;:lp
:~help~%s~::rmail Joe
```

All messages not handled by one of these commands will be delivered to your mailbox as usual.

To install **`mailproc`** processing, you run the command

```
mailproc -y
```

mailx

This section introduces the **`mailx`** facility. It explains how to set up your **`mailx`** environment, send messages with the **`mailx`** command, and handle messages that have been sent to you. The material is presented in four parts:

- **`mailx`** Overview
- Sending Messages
- Managing Incoming Mail
- The **`.mailrc`** File.

mailx Overview

The **mailx(1)** command is an enhanced version of the **mail(1)** command. There are many options to **mailx** that are not available in **mail** for sending and reading mail. For example, you can define an alias for a single login or for a group. This allows you to send **mail** to an individual using a name or word other than their login ID, and to send **mail** to a whole group of people using a single name or word. Or you can create a multipart message, or create a message which uses enhanced text features such as bold or italics. When you use **mailx** to read incoming mail you can save it in various files, edit it, forward it to someone else, respond to the person who originated the message, and so forth. By using **mailx** environment variables you can develop an environment to suit your individual tastes.

If you type the **mailx** command with one or more logins as arguments, **mailx** decides you are sending mail to the named users, prompts you for a summary of the subject, and then waits for you to type in your message or issue a command. The section *"How to Send Messages"* describes features that are available to you for editing, incorporating other files, adding names to copy lists, and more.

If you enter the **mailx** command with no arguments, **mailx** checks incoming mail for you in a file named **/var/mail/***your_login*. If there is mail for you in that file, you are shown a list of the items and given the opportunity to read, store, remove or transfer each one to another file. The section entitled *"How to Manage Incoming Mail"* provides some examples and describes the options available.

If you choose to customize **mailx**, you should create a start-up file in your home directory called **.mailrc**. The section on *"The .mailrc File"* describes variables you can include in your start-up file.

mailx has two modes of functioning: input mode and command mode. You must be in input mode to create and send messages. Command mode is used to read incoming mail. You can use any of the following methods to control the way **mailx** works for you:

- by entering options on the command line. (See the **mailx(1)** manual page in the online *Command Reference.)*
- by issuing commands when you are in input mode, for example, creating a message to send. These commands are normally preceded by a tilde (~) and are referred to as tilde escapes. (See the **mailx(1)** manual page in the online *Command Reference.)*
- by issuing commands when you are in command mode, for example, reading incoming mail.
- by storing commands and environment variables in a start-up file in your home directory called **$HOME/.mailrc**.

Tilde escapes are discussed in *"How to Send Messages: The Tilde Escapes,"* command mode commands in *"How to Manage Incoming Mail,"* and the **.mailrc** file in *"The .mailrc File."*

Command Line Options

In this section, we will look at command line options.

The syntax for the **mailx** command is:

mailx [*options*] [*name* . . .] <RETURN>

The *options* are flags that control the action of the command, and *name*. . . represents the intended recipients.

Anything on the command line other than an option preceded by a hyphen is read by **mailx** as a *name*; for example, the login or alias of a person to whom you are sending a message.

One valuable command line option that is also available in **mail**, is

-f [*filename*]:	Allows you to read messages from *filename* instead of your mailbox. Because **mailx** lets you store messages in any file you name, you may use the **-f** option to review these stored messages. The default storage file is **$HOME/mbox**, so the command **mailx -f** <RETURN> is used to review messages stored there.

How to Send Messages: The Tilde Escapes

To send a message to another UNIX system user, enter the following command:

```
$ mailx login <RETURN>
Subject:
```

The login name specified belongs to the person who is to receive the message. The system puts you into input mode and prompts you for the subject of the message. (You may have to wait a few seconds for the `Subject:` prompt if the system is very busy.) This is the simplest way to run the **mailx** command; it differs little from the way you run the **mail** command.

The following examples show how you can edit messages you are sending, incorporate existing text into your messages, change the header information, and do other tasks that take advantage of the **mailx** command's capabilities. Each example is followed by an explanation of the key points illustrated in the example.

```
$ mailx sms <RETURN>
Subject:
```

Whether to include a subject or not is optional. If you elect not to, press the <RETURN> key. The cursor moves to the next line and the program waits for you to enter the text of the message.

```
$ mailx sms <RETURN>
Subject: meeting notice <RETURN>
We're having a meeting for novice mailx users in <RETURN>
the auditorium at 9:00 tomorrow. <RETURN>
Would you be willing to give a demonstration? <RETURN>
Bob <RETURN>
<CTRL><d> EOT
$
```

There are two important things to notice about the above example:

- You break up the lines of your message by pressing the <RETURN> key at the end of each line. This makes it easier for the recipient to read the message, and prevents you from overflowing the line buffer.
- You end the text and send the message by entering a tilde, a period and a <RETURN> together (`\~.`), or a <CTRL><d>, at the beginning of a line. (If the proper option is set, the message may be ended by entering a period and a <RETURN> together (`.`) at the beginning of a line.) The system responds with an end-of-text notice (EOT) and a prompt.

There are several commands available to you when you are in input mode. Each of the options consists of a tilde (\~), followed by an alphabetic character, entered at the beginning of a line. Together they are known as tilde escapes. (See the **`mailx(1)`** manual page in the online *Command Reference.)* Most of them are used in the examples in this section.

You can include the subject of your message on the command line by using the **`-s`** option. For example, the command line:

```
$ mailx -s "meeting notice" sms <RETURN>
```

is equivalent to:

```
$ mailx sms <RETURN>
Subject: meeting notice <RETURN>
```

The subject line will look the same to the recipient of the message. Notice that when putting the subject on the command line, you must enclose a subject that has more than one word in quotation marks.

Editing the Message

When you are in the input mode of **`mailx`**, you can invoke an editor by entering the `~e` (tilde e) escape at the beginning of a line. The following example shows how to use tilde:

```
$ mailx sms <RETURN>
Subject: Testing my tilde <RETURN>
When entering the text of a message <RETURN>
that has somehow gotten grabled <RETURN>
you may invoke your favorite editor <RETURN>
by means of a <tilde e> (~e). <RETURN>
    .
    .
    .
```

Notice that you have misspelled a word in your message. To correct the error, use `\~e` to invoke the editor, in this case the default editor, **`ed(1).`**

```
    .
    .
    .
\~e <RETURN>
132
/grabled/p <RETURN>
that has somehow gotten grabled
s/gra/gar/p <RETURN>
that has somehow gotten garbled
w <RETURN>
132
q <RETURN>
(continue)
What more can I tell you? <RETURN>
    .
    .
    .
```

In this example the **`ed`** editor was used. The environment variable `$EDITOR` or a **`.mailrc`** file controls which editor will be invoked when you issue a **`\~e`** escape command. The **`\~v`** (tilde v) escape invokes an alternate editor (most commonly, **`vi`**) as specified by the environment variable `$VISUAL`.

When you exited from **`ed`** (by typing `q`), the **`mailx`** command returned you to input mode and prompted you to continue your message. At this point you may want to preview your corrected message by entering a `\~p` (tilde p) escape. The `\~p` escape prints out the entire message up to the point where the `\~p` was entered. Thus, at any time during text entry, you can review the current contents of your message.

```
    .
    .
    .
\~p <RETURN>
Message contains:
To: sms
Subject: Testing my tilde

When entering the text of a message
that has somehow gotten garbled
you may invoke your favorite editor
by means of a <tilde e> (~e).
What more can I tell you?
(continue)
~. <RETURN>
EOT
$
```

Incorporating Existing Text into Your Message

`mailx` provides four ways to incorporate material from another source into the message you are creating. You can:

- read a file into your message
- read a message you have received into a reply
- incorporate the value of a named environment variable into a message
- execute a shell command and incorporate the output of the command into a message.

The following examples show the first two of these functions. These are the most commonly used of these four functions. For information about the other two, see the **`mailx(1)`** manual page of the online *Command Reference.*

Reading a File into a Message

```
$ mailx sms <RETURN>
Subject: Work Schedule <RETURN>
As you can see from the following <RETURN>
\~r letters/file1 <RETURN>
“letters/file1”   10/725
we have our work cut out for us. <RETURN>
Please give me your thoughts on this. <RETURN>
- Bob <RETURN>
~. <RETURN>
EOT
$
```

As the example shows, the `\~r` (tilde r) escape is followed by the name of the file you want to include. The system displays the file name and the number of lines and characters it contains. You are still in input mode and can continue with the rest of the message.

When the recipient gets the message, the text of **`letters/file1`** is included. (You can, of course, use the `\~p` (tilde p) escape to preview the contents before sending your message.)

Incorporating a Message from Your Mailbox into a Reply

```
$ mailx <RETURN>
mailx version 4.0  Type ? for help.
"/var/mail/roberts": 2 messages 1 new
>N   1 abc        Mon Apr 30 16:57  8/155  Meeting Notice
     2 hqtrs      Tue May 1  08:09  4/127  Schedule
? m jones <RETURN>
Subject: Hq Schedule <RETURN>
Here is a copy of the schedule from headquarters... <RETURN>
\~f 2 <RETURN>
Interpolating: 2
(continue)
As you can see, the boss will be visiting our district on <RETURN>
the 14th and 15th. <RETURN>
- Robert <RETURN>
~. <RETURN>
EOT
?
```

There are several important points illustrated in this example:

- The sequence begins in command mode, where you read and respond to your incoming mail. Then you switch into input mode by issuing the command **`\m jones`** (meaning send a message to `jones`).
- The `\~f` escape is used in input mode to forward a message in your mailbox and make it part of the outgoing message. The number `2` after the `~f` means message 2 is to be interpolated (read in).
- **`mailx`** tells you that message 2 is being interpolated and then tells you to continue.
- When you finish creating and sending the message, you are back in command mode, shown by the `?` prompt. You may now do something else in command mode, or quit **`mailx`** by typing `q`.

An alternate command, the **`\~m`** (tilde m) escape, works the way that `\~f` does except the read-in message is indented one tab stop. Both the **`\~m`** and **`\~f`** commands work only if you start out in command mode and then enter a command that puts you into input mode. Other commands that work this way will be covered in the section *"How to Manage Incoming Mail."*

Sending Enhanced Text

`mailx` also supports the creation of enhanced text, in particular, it allows you to create mail which uses the **`text/enriched`** format. For example, mail can be created which includes words in bold or *italic*. When you read the mail message, both **`mail`** and **`mailx`** invoke a program known as **`metamail`** to interpret the enhanced mail. **`metamail`** invokes other programs, as necessary, to present the enhanced mail to you on the screen in a suitable fashion. For example, you can send yourself some enhanced mail, and then read it. The bold and italic words will be presented to you in another font to distinguish them from the rest of the text.

```
$ mailx your-id <RETURN>
Subject: enhanced text <RETURN>
Here is some mail using <RETURN>
\~Tb <RETURN>
Beginning bold
some bold words <RETURN>
\~Tb <RETURN>
and
\~Ti <RETURN>
some italic words. <RETURN>
How do you like it? <RETURN>
~. <RETURN>
EOT
$ mail <RETURN>
Subject: enhanced text

--- invoking richtext
Here is some mail using some bold words and some italic
words. How do you like it?
$
```

`mailx` provides other tilde escapes for creating **`text/enriched`** mail. Typing the help tilde escape `\~?` will show you the complete list.

`mailx` also has the capability of adding attachments to your mail message by using the `\~**` tilde escape.

```
$ mailx roger <RETURN>
Subject: spread sheet <RETURN>
I'm sending you the latest figures on the merger.  Let <RETURN>
me know what you think. <RETURN>
-- Sue <RETURN>
~**
If you want to include non-textual data from a file, enter the file name.
If not, just press ENTER (<RETURN>): figures.lot <RETURN>
Please choose which kind of data you wish to insert:

0: A raw file, possibly binary, of no particular data type.
1: Raw data from a file, with you specifying the content-type by hand.
Enter your choice as a number from 0 to 1: 0 <RETURN>
Type a description to include with the file, <RETURN> to skip: latest figures
<RETURN>
Included data in 'application/octet-stream' format
~. <RETURN>
EOT
$ mail <RETURN>
From: sue@your.company.com
To: roger
Subject: spread sheet

I'm sending you the latest figures on the merger.  Let
me know what you think.
-- Sue
Content-Description: latest figures

This message contains raw digital data, which can either be viewed as text
or written to a file.

What do you want to do with the raw data?
1 -- See it as text
2 -- Write it to a file
3 -- Just skip it

2 <RETURN>
Please enter the name of a file to which the data should be written
(Default: /tmp/mm.a0056l) > figures.lot <RETURN>
Wrote file figures.lot
?
```

Changing Parts of the Message Header

The header of a **`mailx`** message has four components:

- subject
- recipient(s)
- copy-to list
- blind-copy list (a list of intended recipients that is not shown on the copies sent to other recipients).

When you enter the **`mailx`** command followed by a login or an alias you are put into input mode and prompted for the subject of your message. Once you end the subject line by pressing the `<RETURN>` key, **`mailx`** expects you to type the text of the message. If, at any point in input mode, you want to change or supplement some of the header information, there are five tilde escapes that you can use: `\~h`, `\~t`, `\~c`, `\~b` and `\~s`.

`\~h`	displays all the header fields: subject, recipient, copy-to list, and blind copy list, with their current values. You can change a current value, add to it, or, by pressing the `<RETURN>` key, accept it.
`\~t`	lets you add names to the list of recipients. Names can be either login names or aliases.
`\~c`	lets you create or add to a copy-to list for the message. Enter either login names or aliases of those to whom a copy of the message should be sent.
`\~b`	lets you create or add to a blind-copy list for the message.
`\~s`	lets you change the subject line for the message.

All tilde escapes must be in the first position on a line. For the `\~t`, `\~c` or `\~b`, any additional material on the line is taken to be input for the list in question. The argument for `\~s` replaces the current subject. Entering the `\~h` tilde escape will display each of the header lines, allowing you to backspace and make changes. Any additional material on the `\~h` line is ignored.

Adding Your Signature

If you want, you can establish two different signatures with the `sign` and `Sign` environment variables. These can be invoked with the `\~a` (tilde a) or `\~A` (tilde A) escape, respectively. Assume you have set the value of the `Sign` variable to `Supreme Commander` to be called by the `\~A` escape. Here's how it would work:

```
$ mailx -s orders bll <RETURN>
Be ready to move out at 0400 hours. <RETURN>
~A <RETURN>
Supreme Commander
~. <RETURN>
EOT
$
```

Having both escapes (`\~a` and `\~A`) allows you to set up two forms for your signature. However, because the sender's login automatically appears in the message header when the message is read, no signature is required to identify you.

Keeping a Record of Messages You Send

The **`mailx`** command offers several ways to keep copies of outgoing messages. Two that you can use without setting any special environment variables are the `\~w` (tilde w) escape and the **`-F`** option on the command line.

The `\~w` followed by a file name causes the text of the message to be written to the named file (the file must not already exist.) For example:

```
$ mailx bdr <RETURN>
Subject: Saving Copies <RETURN>
When you want to save a copy of <RETURN>
the text of a message, use the tilde w. <RETURN>
\~w savemail <RETURN>
“savemail” 2/71
~. <RETURN>
EOT
$
```

If you now display the contents of **`savemail`**, you will see this:

```
$ cat savemail <RETURN>
When you want to save a copy of
the text of a message, use the tilde w.
$
```

The drawback to this method, as you can see, is that none of the header information is saved.

The **`-F`** option appends the text of the message to a file named after the first recipient. If you have used an alias for the recipient(s) the alias is first converted into the appropriate login(s) and the first login is used as the file name. If you have a file by that name in your current directory, the text of the message is appended to it.

Using the **`-F`** option on the command line does preserve the header information. It works as follows:

```
$ mailx -F bdr <RETURN>
Subject:  Savings <RETURN>
This method appends this message to a <RETURN>
file in my current directory named bdr. <RETURN>
~. <RETURN>
EOT
$
```

We can check the results by looking at the file **`bdr`**.

```
$ cat bdr <RETURN>
From kol Fri May  2 11:14 EST 1993
To: bdr
Subject: Savings

This method appends this message to a
file in my current directory named bdr.
$
```

Exiting from mailx

When you have finished composing your message, you can leave **`mailx`** by typing any of the following three commands:

`\~.` a tilde and a period (**`\~.`**) followed by `<RETURN>`, or `<CTRL><d>`, are the standard ways of leaving input mode. They also send the message. (If the proper option is set, the message may also be ended by entering a period and a `<RETURN>` together (**`.`**) at the beginning of a line.) If you entered input mode from the command mode of **`mailx`**, you now return to the command mode (shown by the `?` prompt you receive after typing this command). If you started out in input mode, you now return to the shell (shown by the shell prompt).

`\~q` tilde q (**`\~q`**) quits sending the mail message. If you have entered text for a message, it will be appended to the file called **`dead.letter`** in your home directory.

`\~x` tilde x (**`\~x`**) quits sending the mail message without saving anything.

Summary

In the preceding paragraphs we have described and shown examples of some of the tilde escape commands available when sending messages via the **`mailx`** command. (See the **`mailx(1)`** manual page in the online *Command Reference.)*

How to Manage Incoming Mail

`mailx` has over fifty commands that help you manage your incoming mail. See the **`mailx(1)`** manual page in the online *Command Reference* for a list of all of them (and their synonyms) in alphabetic order. The most commonly used commands (and arguments) are described in the following subsections:

- the *msglist* argument
- commands for reading and deleting mail
- commands for saving mail

- commands for replying to mail
- commands for getting out of **`mailx.`**

The msglist Argument

Many commands in **`mailx`** take a *msglist* argument. This argument provides a command with a list of messages on which to operate. If a command expects a *msglist* argument and you do not provide one, the command is performed on the current message. Any of the following formats can be used for a *msglist*:

`.`	the current message
n	message number *n*
`^`	the first undeleted message
`$`	the last undeleted message
`*`	all undeleted messages
n-m	an inclusive range of message numbers
user	all messages from *user*
/*string*	All messages with *string* in the subject line (case is ignored)
`:`*c*	all messages of type *c* where *c* is:

```
d - deleted messages
n - new messages
o - old messages
r - read messages
s - saved messages
u - unread messages
```

The context of the command determines whether this type of specification makes sense. For the **`undelete`** command, the *msglist* `^`, `$` and **`*`** commands refer to deleted messages.

Here are two examples (the `?` is the command mode prompt). The first command deletes messages 1, 2 and 3. The second command saves all messages from user `bdr` in a file named **`bdrmail`**.

```
? d 1-3 <RETURN>
? s bdr bdrmail <RETURN>
?
```

Additional examples may be found throughout the next three subsections.

Commands for Reading and Deleting Mail

When a message arrives in your mailbox, the following notice appears on your screen:

```
you have mail
```

The notice appears when you log in or when you return to the shell from another procedure.

Reading Mail

To read your mail, enter the **`mailx`** command without arguments. Execution of the command places you in the command mode of **`mailx`**. Your screen will look something like this:

```
mailx version 4.0  Type ? for help.
“/var/mail/bdr”:   3 messages  3 new
> N 1 rbt         Thu  Apr 30 14:20   8/190  Review Session
  N 2 admin       Thu  Apr 30 15:56   5/84   New printer
  N 3 sms         Fri  May  1 08:39  64/1574 Reorganization
?
```

The first line identifies the version of **`mailx`** used on your system, and reminds you that help is available by typing a question mark (`?`). The second line shows the path name of the file used as input to the display (the file name is normally the same as your login name) together with a count of the total number of messages and their status. The rest of the display is header information from the incoming messages. The messages are numbered in sequence with the last one received having the highest number. To the left of the numbers there may be a status indicator; N for new, U for unread. A greater than sign (`>`) points to the current message. Other fields in the header line show the login of the originator of the message, the day, date and time it was sent, the number of lines and characters in the message, and the message subject. The last field may be blank.

When the header information is displayed on your screen, you can print messages either by pressing the `<RETURN>` key or entering a command followed by a *msglist* argument. If you enter a command with no *msglist* argument, the command acts on the message pointed to by the greater than sign (`>`). Pressing the `<RETURN>` key is equivalent to typing the **`n`** (for *next*) command. Typing the **`p`** (for *print*) command without a *msglist* argument displays the message pointed at by the `>` sign. To read some other message (or several others in succession), enter a **`p`** (for print) or **`t`** (for type) followed by the message number(s). (The command **`t`** (for type) is a synonym of **`p`** (for print).) Here are some examples, which print the current message, print message number 2, and print all messages from user `sms`.

```
?  <RETURN>
? p 2 <RETURN>
? p sms <RETURN>
```

Scanning Your Mailbox

The **`mailx`** command lets you look through the messages in your mailbox while you decide which ones need your immediate attention.

When you first enter the **`mailx`** command mode, the banner tells you how many messages you have and displays the header lines for up to twenty messages. If you are connected to the computer over a slow communication line, only the header lines for up to five (below 1200 baud) or ten messages (at 1200 baud) are displayed. If the total number of messages exceeds one screenful, you can display the next screen by entering the **`z`** command. Typing `z-` causes the previous screen (if there is one) to be displayed. If you want to see the header information for a specific group of messages, enter the **`f`** (for from) command followed by the *msglist* argument.

Here are examples of those commands, which scroll forward one screenful of header lines, scroll backward one screenful, and display headers of all messages from user `sms`.

```
? z <RETURN>
? z- <RETURN>
? f sms <RETURN>
```

Switching to Other Mail Files

When you enter **`mailx`** by issuing the command:

```
$ mailx <RETURN>
```

you are looking at the file **`/var/mail/`***your_login*.

`mailx` lets you switch to other mail files and use any of the **`mailx`** commands on their contents. (You can even switch to a non-mail file, but if you try to use **`mailx`** commands you are told `No applicable messages`.) The switch to another file is done with the **`fi`** or **`fold`** command (they are synonyms) followed by the *filename*. The following special characters work in place of the *filename* argument:

`%`	your default mailbox (**`/var/mail/`***your_login*)
`%`*login*	the mailbox of the owner of *login* (if you have the required permissions)
`#`	the previous file
`&`	your mbox

Here is an example of how this might look on your screen:

```
$ mailx <RETURN>
mailx version 4.0  Type ? for help.
"/var/mail/sms":  3 messages 2 new 3 unread
  U 1 jaf         Sat May 9 07:55   7/137   test25
> N 2 todd        Sat May 9 08:59   9/377   UNITS requirements
  N 3 has         Sat May 9 11:08  29/1214  access to bailey? fi & <RETURN>
    [ Enter this command to transfer to your mbox. ] Held 3 messages in /var/
mail/sms
"+mbox":  74 messages 10 unread
  .      [ Enter any commands for your mbox. ]
  .
  .
? q <RETURN>
$
```

Deleting Mail

To delete a message, enter a `d` followed by a *msglist* argument. If the *msglist* argument is omitted, the current message is deleted. The messages are not deleted until you leave the mailbox file you are processing. Until you do, the `u` (for `u`ndelete) gives you the opportunity to change your mind. Once you have issued the quit command (**`q`**) or switched to another file, however, the deleted messages are gone.

`mailx` permits you to combine the delete and print command and enter a **`dp`**. This is like saying, "Delete the message I just read and show me the next one." Here are some examples of the delete command: delete all my messages, delete all messages that have been read, delete the current message and print the next one, and delete messages 2 through 5.

```
? d * <RETURN>
? d :r <RETURN>
? dp <RETURN>
? d 2-5 <RETURN>
```

Commands for Saving Mail

All messages not specifically deleted are saved when you quit **`mailx`**. Messages that have been read are saved in a file in your home directory called **`mbox`**. Messages that have not been read are held in your mailbox (**`/var/mail/`***your_login*).

The command to save messages comes in two forms: with an upper case or a lower case `s`. The syntax for the upper case version is:

`S [`*msglist*`]`

Messages specified by the *msglist* argument are saved in a file in the current directory named for the login of the first message in the list.

The syntax for the lower case version is:

`s` [*msglist*] *filename* *or*
`s`

Messages specified by the *msglist* argument are saved in the file named in the *filename* argument. If you omit the *msglist* argument, the current message is saved. Finally, if both the *msglist* and the *filename* are omitted, the mail is saved in a file called **`mbox`** in your home directory.

Commands for Replying to Mail

The command for replying to mail comes in two forms: with an upper case or a lower case `r`. The difference between the two forms is that the upper case form (**`R`**) causes your response to be sent only to the originator of the message, while the lower case form (**`r`**) causes your response to be sent not only to the originator but also to all other recipients.

When you reply to a message, the original subject line is picked up and used as the subject of your reply. Here's an example of the way it looks:

```
$ mailx <RETURN>
mailx version 4.0  Type ? for help.
“/var/mail/sms”:  3 messages 2 new 3 unread
  U 1 jaf         Wed May 9 07:55   7/137   test25
> N 2 todd        Wed May 9 08:59   9/377   UNITS requirements
  N 3 has         Wed May 9 11:08  29/1214  access to bailey
? R 2 <RETURN>
To: todd
Subject: Re: UNITS requirements
```

Assuming the message about “UNITS requirements” had been sent to some additional people, and the lower case `r` had been used, the header might have appeared like this:

```
? r 2 <RETURN>
To: todd, eg, has, jcb, bdr
Subject: Re: UNITS requirements
```

Commands for Getting out of mailx

There are two standard ways of leaving **`mailx`**: with a **`q`** or with an **`x`**. If you leave **`mailx`** with a **`q`**, you see messages that summarize what you did with your mail. They look like this:

```
? q <RETURN>
Saved 1 message in /fs1/bdr/mbox
Held 1 message in /var/mail/bdr
$
```

From the example, we can surmise that user `bdr` had at least two messages, read one and either left the other unread or issued a command asking that it be held in **`/var/mail/bdr`**. If there were more than two messages, the others were deleted or saved in other files.

If you leave **`mailx`** with an **`x`**, it is almost as if you had never entered. Mail read and messages deleted are retained in your mailbox. However, if you have saved messages in other files, that action has already taken place and is not undone by the **`x`**.

mailx Command Summary

In the preceding subsections we have described some of the most frequently used **`mailx`** commands. (See the **`mailx(1)`** manual page in the online *Command Reference* for a complete list.) If you need help while you are in the command mode of **`mailx`**, type either a `?` or `help` at the `?` prompt. A list of **`mailx`** commands and what they do will be displayed on your terminal screen.

The .mailrc File

The **`.mailrc`** file contains commands to be executed when you invoke **`mailx`**.

There may be a system-wide start-up file (**`/etc/mail/mailx.rc`**) on your system. If it exists, it is used by the system administrator to set common variables. Variables set in your **`.mailrc`** file take precedence over those in **`mailx.rc.`**

Most **`mailx`** commands are legal in the **`.mailrc`** file. However, the following commands are *not* legal entries:

`!` (or) **`shell`**	escape to the shell
`Copy`	save messages in *msglist* in a file whose name is derived from the author
`edit`	invoke the editor on a text message
`bedit`	invoke the editor on a binary message
`visual`	invoke the visual editor on a text message
`bvisual`	invoke the visual editor on a binary message
`followup`	respond to a message

`Followup`	respond to a message, sending a copy to the author of each message in *msglist*
`hold` (or) **`preserve`**	hold (preserve) a message in the mailbox
`mail`	switch into input mode
`reply`	respond to a message
`Reply`	respond to the author of each message in *msglist*

You can create your own **`.mailrc`** with any editor. Screen 10-1 shows a sample **`.mailrc`** file.

```
if r
  cd $HOME/mail
endif
set allnet append asksub askcc autoprint dot
set metoo quiet save showto header hold keep keepsave
set outfolder
set folder='mail'
set record='outbox'
set crt=24
set EDITOR='/bin/ed'
set sign='Roberts'
set Sign='Jackson Roberts, Supervisor'
set toplines=10
alias fred  fjs
alias bob  rcm
alias alice  ap
alias donna  dr
group robertsgrp  fred bob alice mark pat
group accounts  robertsgrp donna
```

Screen 10-1. Sample .mailrc File

The example in Screen 10-1 includes the commands you are most likely to find useful: the **`set`** command and the **`alias`** or **`group`** commands.

The **`set`** command is used to establish values for environment variables. The command syntax is:

> `set`
> `set` *name*
> `set` *name*=*string*
> `set` *name*=*number*
> `set no`*name*

When you issue the **`set`** command without any arguments, **`set`** produces a list of all defined variables and their values. The argument *name* refers to an environmental variable. More than one *name* can be entered after the **`set`** command. Some variables take a string or numeric value. String values are enclosed in single quotes.

When you put a value in an environment variable by making an assignment such as `HOME=`*my_login*, you are telling the shell how to interpret that variable. However, this type of assignment in the shell does not make the value of the variable accessible to other

UNIX system programs that need to reference environment variables. To make it accessible, you must export the variable. If you set the `TERM` variable in your environment, (see Chapter 7, “*Screen Editor (vi) Tutorial*” and Chapter 9, “*Programming with the UNIX System Shell*”) you will remember using the **`export`** command shown in the following example:

```
$ TERM=wyse150
$ export TERM
```

When you export variables from the shell in this way, programs that reference environment variables are said to import them. Some of these variables (such as `EDITOR` and `VISUAL`) are not specific to **`mailx`**, but may be specified as general environment variables and imported from your execution environment. If a value is set in **`.mailrc`** for an imported variable it overrides the imported value. There is an **`unset`** command, but it works only against variables set in **`.mailrc`**; it has no effect on imported variables. Using **`set no`***name* is the same as **`unset`** *name*.

There are too many environment variables that can be defined in your **`.mailrc`** to be fully described in this document. For complete information, consult the **`mailx(1)`** manual page in the online *Command Reference.*

Three variables used in the example in Screen 10-1 deserve special attention because they show how to organize the filing of messages. These variables are: `folder`, `record`, and `outfolder`. All three are interrelated and control the directories and files in which copies of messages are kept.

To put a value into the `folder` variable, use the following format:

`set folder=`*directory*

This specifies the directory in which you want to save standard mail files. If the directory name specified does not begin with a `/` (slash), it is presumed to be relative to `$HOME`. If `folder` is an exported shell variable, you can specify file names (in commands that call for a *filename* argument) with a `/` before the name; the name will be expanded so that the file is put into the `folder` directory.

To put a value in the `record` variable, use the following format:

`set record=`*filename*

This directs **`mailx`** to save a copy of all outgoing messages in the specified file. The header information is saved along with the text of the message. By default, recording is disabled.

The `outfolder` variable causes the file in which you store copies of outgoing messages (enabled by the `record` variable) to be located in the `folder` directory. It is established by being named in a **`set`** command. The default is `nooutfolder`.

The **`alias`** and **`group`** commands are synonyms. In Screen 10-1, the **`alias`** command is used to associate a name with a single login; the **`group`** command is used to specify multiple names that can be referred to by one pseudonym. This is a nice way to distinguish between single and group aliases, but if you want, you can treat the commands as exact equivalents. Notice, too, that aliases can be nested.

In the **`.mailrc`** file shown in Screen 10-1, the alias `robertsgrp` represents five users; four of them are specified by previously defined aliases and one, `mark`, is specified by a login. The next group command in the example, `accounts`, uses the group `robertsgrp` plus the alias `donna`. It expands to six logins.

The **`.mailrc`** file in Screen 10-1 includes an **`if-endif`** command. The full syntax of that command is:

```
if s | r | t
  mail_commands
else
  mail_commands
endif
```

The `s`, `r` and `t` stand for send, receive and terminal. You can cause some initializing commands to be executed according to whether **`mailx`** is entered in input mode (send), in command mode (receive), and whether the input is coming from a terminal. In the preceding example, the command is issued to change directory to **`$HOME/mail`** if reading mail. Here, the user elected to set up a subdirectory to handle incoming mail.

The environment variables shown in this section are those most commonly included in the **`.mailrc`** file. You can, however, specify any of them for one session only whenever you are in command mode. For a complete list of the environment variables you can set in **`mailx`**, see the **`mailx(1)`** manual page in the online *Command Reference.*

Using Mail in a Secure Environment

In a secure environment with the Enhanced Security package installed, mail is run at multiple levels. See Chapter 14, "*Managing Files Securely*" for a complete explanation of security levels.

All mail is sent and received within a security level; mail never crosses security levels. Mail sent from a user logged in at one security level will arrive to the other user at the same security level. In order for that other user to read and manipulate that mail message, they must also log in to that security level. This is all made possible through the use of a Multi-Level Directory (MLD) for **`/var/mail`**. Even though it is not normally visible, there actually are separate instances of **`/var/mail`** at each of the different security levels.

Checking for Mail: The mailcheck Command

Since the shell normally only informs you of mail arriving at the security level at which you are logged in, a command is provided, **`mailcheck`**, which checks to see if mail exists at other security levels. Note that it only checks for mail at levels dominated by the current login security level. A useful addition to your **`$HOME/.profile`** would be:

```
mailcheck 2>/dev/null
```

This informs you of mail when you log in, but says nothing if there is no mail. For example, you might see the message

```
You have mail at level: Top Secret
You have mail at level: Unclassified
```

Note that to check to see if you have mail at all levels at which you can log in, you must log in at your *highest* security level, the level that dominates all other valid login levels for your login, and run **`mailcheck`**.

Saving Mail in Multiple Levels

Another consequence of having mail at different levels is that you will not be able to save the mail from different levels into the same file. By default, the mail programs will save mail into the file **`$HOME/mbox`**. Saving mail into this file from one level will prevent that file from being written to from another level. One solution is to choose a different filename for different security levels when saving mail. A simpler solution is to make **`$HOME/mbox`** into a Multi-Level Directory using the command

```
mkdir -M $HOME/mbox
```

The mail command will automatically use the filename **`$HOME/mbox/mbox`** if it finds that **`$HOME/mbox`** is a directory. In this fashion, there will be a separate mbox file for each security level.

Getting around the System

Note that it is still possible to read mail messages at dominated security levels by using *real* mode. Once real mode has been set, all dominated mail boxes will be accessible directly as regular files and may be viewed using such commands as **`pg`**. Note that you will *not* be able to make any changes to any mail files not at your current level.

Mail Forwarding, Vacation and notify

Because there is a separate mailbox for each security level at which you can log in, mail forwarding processing must be established separately for each level. Because of this, you can use different forwarding lists for each security levels; the people to whom mail should be forwarded *Top Secret* will not necessarily be the same people to whom mail should be forwarded at *Unclassified.*

Forwarding processing affects not only forwarding mail to other users, but also the programs **`vacation`** and **`notify`**. If you go on vacation and wish to use the **`vacation`** program, you will have to log into each of your accessible security levels and run the **`vacation`** at each level.

11
Remote Services Tutorial

11
Remote Services Tutorial

Introduction

The UNIX system includes various user programs that enable you to perform operations on remote hosts using the TCP/IP networking software. This chapter covers many of the most frequently used commands.

This chapter is not intended to be a replacement for the remote services manual pages. It does not cover all user level network commands, nor does it discuss all applications of the commands that are covered.

Remote commands are used with the TCP/IP networking software, which is part of the UNIX system. TCP/IP must be running for any of the commands to work.

If you have any trouble with these commands, or if TCP/IP is not running and you want to use these commands, talk to your system administrator.

Overview of Remote Commands

A remote command is a command which runs on a different machine than the one you are using. Remote commands allow you to access machines of different architectures, or even machines which are not running the same operating system.

User commands such as those in the following table

Table 11-1. User Commands

`rlogin`	(remote login)
`rsh`	(remote shell)
`rcp`	(remote copy)
`ftp`	(file transfer program)

let you log in, create and use a shell, get information, and copy files on a remote machine. The **`finger`** command is useful to display information on any user you specify.

Before you can use it, **`finger`** service should be enabled must be enabled on the remote machine. You can then use the **`telnet`** command to log in to other machines, whether or not they run the UNIX operating system.

Terminology

In reading the description of some of the commands, you may encounter terms which are unfamiliar to you. Most of the new terms used in this chapter are explained here. If you find a term with which you are unfamiliar and which is not listed here, ask your system administrator.

abort	Discontinuing a process in the middle without waiting for the normal exit. Aborting is normally achieved by sending an interrupt signal to the program you are running.
daemon	A program that handles jobs automatically. Many remote commands get information from a remote machine by exchanging data with daemons running on the remote machine. An example of a daemon is the mail service, which processes mail automatically and routes it to the intended recipient.
local	The machine you are currently logged in to. It is contrasted with the term “remote” machine (see below).
remote	The machine to which you are connected across the network. You interact with the remote machine by using the commands shown in this chapter.
shell	The program that you use to interface with your machine. A local shell is the shell you are running on the local machine, and a remote shell is the shell that runs on the machine to which you are connected. The Bourne Shell normally generates a dollar sign ($) prompt to show that it is ready to accept a command.
suspend	To halt temporarily whatever program you are running. You can return to a suspended program at any time and resume exactly where you left off.

Conventions

In this chapter, prompts in screens are shown in the form *hostname*`$`, to make clear whether the shell is a local or a remote shell. You can set your terminal to include the host name in the prompt by adding the following line to your **`$HOME/.profile`** file:

```
PS1=“`hostname`$PS1”
```

Copying Files between Machines

The **`rcp`** program lets you copy files from your machine to another one, and vice versa. The basic syntax of **`rcp`** is:

`rcp` *source destination*

where the *source* is where the file is coming from, and *destination* is where it is going. The specific form that the two arguments take for different types of transfers are shown in the following sections.

Copying from Another Machine to Your Machine Using rcp

To copy a file from another machine to your machine with **`rcp`**, use the following syntax:

`rcp` *machinename:file directory*

where *machinename* is the name of the machine you want to copy from; *file* is the file you want to copy; and *directory* is where you want to put the file on your system.

For example, to copy a file called **`/home/charon/new.toy`** from the machine called `pluto` to the directory called **`/home/medici/toys`** on your machine, `venus`, type

```
rcp pluto:/home/charon/new.toy /home/medici/toys
```

You can use normal shorthand for directories (such as `$HOME` for your home directory when using the Bourne shell, `.` for the current directory and `..` for the parent directory).

When you want to call the file by a different name on your own machine, specify a destination *filename* at the end of the destination directory on your machine. For example, you could copy the file **`new.toy`** from machine `pluto` to your home directory and rename it **`my.toy`** by typing

```
rcp pluto:/home/charon/new.toy $HOME/my.toy
```

Copying from Your Machine to Another Machine Using rcp

To **`rcp`** a file from your machine onto another machine, reverse the syntax described in the preceding section:

`rcp` *file machinename:directory*

where *file* is the file on your machine you want to copy; *machinename* is name of the machine you want to copy to; and *directory* is the place you want to send the file to.

For example, to copy a file called **`/home/medici/old.toy`** from your machine to the directory called **`/home/charon/trash`** on the machine `pluto` type

```
rcp /home/medici/old.toy pluto:/home/charon/trash
```

When you want to call the file by a different name on the other machine, specify a destination file at the end of the destination directory on that machine. For example, typing

```
rcp /home/medici/old.toy pluto:/home/charon/trash/toy
```

will copy the file **`old.toy`** from `medici's` home directory to the file named **`toy`** in the directory **`/home/charon/trash`** in the machine `pluto.`

Copying Directories with rcp

To copy a directory and its contents from another machine to your machine, or vice versa, use **`rcp`** with the **`-r`** option. Then, follow the steps for copying files; replace the filenames with the appropriate directory names.

NOTE

Copying directories with **`rcp`** doesn't preserve ownership settings, nor does it necessarily preserve permissions.

To copy a directory and its contents from another machine to your machine, the syntax is

`rcp` **`-r`** *machinename*`:`*directory local_directory*

where *machinename* is the name of the remote machine; *directory* is the directory on that machine that you want to copy; and *local_directory* is the directory on your machine you want to copy to.

To copy a directory and its contents from your machine to another machine, the syntax is

`rcp` **`-r`** *local_directory machinename*`:`*directory*

where *local_directory* is the directory on your machine you want to copy, and *directory* is the place on the other machine you want to copy to.

Expanding Shell Metacharacters During rcp

Any shell metacharacters that are not escaped or quoted result in their expansion at the local level, not at the level of the remote machine.

This applies also to the redirection characters, `>`, `<`, and `|`.

Error Messages

An error message you commonly get when trying to do a remote copy is

```
...Permissiondenied”
```

This error message may indicate that

- You do not have read permission on the file you want to copy.
- You do not have write permission on the directory you want to copy to.
- You do not have permission to access files in the remote machine because your machine's name is not in the remote machine's list of trusted hosts.

If you receive the message

```
Login incorrect
```

you do not have permission to access files in the remote machine because your name is not in that machine's password database.

In all cases when you receive an error message, consult with your system administrator.

Executing Commands Remotely

The **`rsh`** command allows you to execute a single command on another machine without having to log in formally (**`rsh`** stands for `r`emote `sh`ell, that is, an interpreter capable of executing commands on another machine). **`rsh`** can save time when you know you only want to do one thing on the remote machine.

To execute a command on another machine, type **`rsh`** followed by the machine's name and the command. For example, if you want to see the contents of the directory **`/home/fresno/crops`** on the machine `fresno`, type

```
rsh fresno ls -C /home/fresno/crops
```

When you execute a command on another machine using **`rsh`**, **`rsh`** doesn't log in; it talks to a daemon that spawns a shell for you and executes the command on the other machine. The type of shell spawned depends on the configuration of the entry for you in the remote machine's password database.

Like **`rlogin`** and **`rcp,rsh`** uses the other machine's password database and the files **`/etc/hosts.equiv`** and **`.rhosts`** to determine whether you have unchallenged access privileges.

Expanding Shell Metacharacters During rsh

As in the case of **`rcp,`** any shell metacharacters that are not escaped or quoted result in their expansion at the local level, not at the level of the remote machine.

This applies also to the redirection characters, `>`, `<`, and **`|.`** For instance, if you were to enter on machine `oak` the command

```
rsh willow ls /etc > /tmp/list
```

the output of the **`ls`** command on machine `willow` would be redirected to a file **`/tmp/list`** on machine `oak;` but if you enter the command

```
rsh willow ls /etc '>' /tmp/list
```

the output would now be redirected to a file **`/tmp/list`** on machine `willow,` because the redirection would not happen now at the local level.

Calling rsh with No Commands

If you call **rsh** using the syntax

```
rsh machinename
```

that is, with no arguments after the name of the remote machine, **rsh** will behave exactly as if you had entered

```
rlogin machinename
```

and you will be logged in at the remote machine (assuming you have permission).

Calling rsh by a Different Name

The command **rsh** can be called under a different name, by making a symbolic link between the file **/usr/bin/rsh** and a file called by the name of the remote host.

For example, to create a symbolic link between **rsh** and a remote host called `willow`, you would enter the command

```
ln -s /usr/bin/rsh /usr/hosts/willow
```

Now, provided the directory **/usr/hosts** is in your search path, you can enter the command

```
willow
```

on your machine to log in to machine `willow.`

If you want to obtain a listing of the directory **/etc** on machine `willow,` you can enter

```
willow ls /etc
```

You can repeat the linking process for all the machines that you frequently access remotely. Note that making the symbolic links in the directory **/usr/hosts** is a convention; you can make them in any directory, as long as you have permission to create files in the directory and it is in your search path.

Logging in on Remote UNIX Machines with rlogin

The **rlogin** command logs you in to other UNIX system machines on a network. To log in to a UNIX system machine on a network, type **rlogin** and the *machine name* of the other machine. If your machine's name is in the other machine's **/etc/hosts.equiv** file, or in the **.rhosts** file in your remote home directory, then the other machine trusts your machine name and won't require you to type your password. Otherwise, a password prompt will appear. Type your password for that machine followed by <RETURN>. If you have an entry in the password database of the other machine, and you entered the correct password, you will be logged into the other machine as if you had just physically logged in to it.

```
venus$ rlogin jupiter
Password: (Here you type your password.)
Last login: Mon Oct 20 00:30:52 from venus
jupiter$ pwd
/home/medici
jupiter$ exit
Connection closed.
venus$
```

Aborting an rlogin Connection

To abort an **`rlogin`** connection, type a tilde character followed by a period (`~.`) at the beginning of a line. The login connection to the other machine aborts, and you find yourself back at your original machine.

NOTE

Usually you abort an **`rlogin`** connection only when you can't terminate the connection using **`exit`** or **`logout`** at the end of the work session.

When you log in to a series of machines, accessing each machine through another machine, and you use `~.` to abort the connection to any of the machines in the series, you return to the machine where you started; all the intermediate connections are severed.

```
venus$ rlogin comet
Last login: Thu Nov 21 05:04:03 from venus
comet% ~. (Sometimes ~ doesn't echo.)
Closed connection.
venus$
```

To disconnect to an intermediate **`rlogin,`** use two tildes followed by a period (`~~.`), as shown in the example below:

```
venus% rlogin comet
comet% rlogin jupiter
jupiter% ~~. (Sometimes ~~ doesn't echo.)
comet%
```

Getting rlogin Access

A remote user who has no entry in the password database of a given machine is automatically denied permission to log in to it.

If a user is in the password database, the user will be asked for a password. If the password matches the one stored in the password database, the user will be granted permission to log in to the machine.

If the machine is listed in **`/etc/hosts.equiv`** on the remote host, or if the user has a **`.rhosts`** file located in his or her remote home directory with the machine name listed in it, the user can log in without first supplying a password.

If you cannot **`rlogin`** to a remote machine because you are asked for a password which the machine identifies as incorrect, see your system administrator.

From time to time you may want to log in as someone else, so that you can fully manipulate files on the remote machine. One example of this would be when you are working on someone else's machine (and using their username) and you want to log in to your own machine as yourself. The **`-l`** option to **`rlogin`** allows you to do this. The format is as follows:

`rlogin` *machinename* **`-l`** *username*

However, a number of restrictions apply to the **`-l`** option; for information, see the **`rlogin(1)`** manual page.

Suspending an rlogin Connection

If you are using a job control shell (`jsh` or `ksh`), you can suspend an **`rlogin`** connection, then return to it later. To do so, type the tilde character (~) followed by `<CTRL><Z>`. The **`rlogin`** connection becomes a stopped process, and you are put back into the machine you logged in from. To reactivate the connection, type `fg`, or `%` followed by the job number of the stopped process (the default job number for `%` is the job you most recently stopped or put in the background).

```
venus$ rlogin animation
Last login: Thu Nov 21 07:07:07 from venus
animation$ ~      (Sometimes ^Z doesn't echo on the screen.)

(1) + Stopped (usn) rlogin animation
venus$ pwd
/home/medici
venus$ fg
rlogin animation     (Type <RETURN> here to get the command prompt.)

animation$ logout
Connection closed.
venus%
```

As is the case with aborting **`rlogin`** with `~~.`, using two tildes and a <`CTRL`><`Z`> will suspend you to an intermediate **`rlogin.`** For example, if from `oak` you **`rlogin`** to `willow` and from there to `cypress,` entering `~.` will bring you back to `oak,` but entering `~~.` will bring you back to `willow.`

Logging in to a Machine Running Another Operating System with telnet

Because you can log in from one UNIX system machine to another using **`rlogin`**, you need to use **`telnet`** only when you want to log in to a machine running a different operating system.

Imagine that you want to log in to machine `tops20,` running the TOPS20™ operating system. To log in to `tops20,` type **`telnet`**, followed by its machine name. **`telnet`** notifies you of the connection with the other machine, then identifies your escape character. Now you log in to the machine as you ordinarily would.

```
venus$ telnet tops20
Trying...
Connected to tops20.
Escape character is '^]'.

Yoyodyne Corp., TOPS-20 Monitor 6.1 (6762)-4
@LOG MEDICI

...
```

NOTE

If you attempt to log in to a machine that isn't a part of your network, **`telnet`** displays a notification and a prompt. Exit from **`telnet`** by typing **`quit`**, or the abbreviation **`q`**.

Suspending a telnet Connection

If you are using a job control shell (`jsh` or `ksh`), you can suspend a **`telnet`** connection, then return to it later. To do so, type the standard escape character (usually <`CTRL`><`]`>), followed by `z` at the **`telnet>`** prompt. The **`telnet`** program becomes a background process. To reactivate the connection, type `fg`, or `%` followed by the job number of the background process (the default job number for `%` is the job you most recently put in the background).

```
venus% telnet tops20
Trying...
Connected to tops20.
Escape character is '^]'.

Yoyodyne Corp., TOPS-20 Monitor 6.1 (6762)-4
@LOG MEDICI

...
@      (Type <CTRL><]> to get telnet> prompt.)  telnet> z

Stopped venus% fg telnet tops20    (Type <RETURN> twice to get command prompt of other system.)

@exit
Connection closed by foreign host.
venus%
```

Aborting a telnet Connection

Just as with **rlogin**, you should abort a **telnet** connection only when you can't terminate the connection using **exit** or **logout** at the end of the work session.

If you have to abort a **telnet** connection, type the **telnet** escape character (usually <CTRL><]>, followed by **quit** at the **telnet>** prompt. The login connection to the other machine aborts, and you find yourself back at your original machine.

NOTE

When you log in to a series of machines, accessing each machine through another machine, and you abort the connection to any of the machines in the series, you return to the machine where you originally started.

```
venus$ telnet tops20
Trying...
Connected to tops20.
Escape character is '^]'.

Yoyodyne Corp., TOPS-20 Monitor 6.1 (6762)-4
@LOG MEDICI

...
@      (Type <CTRL><]> to get telnet> prompt.)
telnet> quit
venus$
```

Transferring Files between Machines with ftp

The **`ftp`** program is used to copy files to and from machines on a network. **`ftp`** is somewhat different from **`rcp`** in that it is not necessary to be a user on the remote machine, nor does the remote machine need to be running the same operating system. This command is also useful when you are trying to transfer files with unknown filenames, as **`ftp`** allows you to list directory contents on remote machines.

When you start **`ftp`**, you are placed in an interactive session with the daemon on the remote machine. The daemon is the part of the **`ftp`** program on the remote machine that handles all that needs to be done on that end. Once you are connected, the daemon reports that the connection is established, and then asks for a login. If you have an entry in the password database on the remote machine, you can just press <RETURN> to take the default answer, which is your own username. When you enter the correct password, you will then be given access to files on that machine. Here is a sample of such a login:

```
venus$ ftp dinger
Connected to dinger.
220 dinger FTP server ready.
Name (dinger:stein):  <RETURN>
331 Password required for stein.
Password: (Here you type your password.)
230 User stein logged in.
ftp>
```

As you can see, the password does not echo on the screen when it is entered. When you want to transfer files between a machine on which you do not have an entry in the password database, files can still be accessed if the machine is set up for “anonymous” **`ftp`**. To set up a machine for anonymous **`ftp`**:

1. Create a login for **`ftp`** in your **`/etc/passwd`** file.
2. Make sure you have the directories and files listed in the following table on your system:

Table 11-2. Directory and File Requirements for ftp

File or Directory	Permissions	Owner
/home/ftp	dr-x--x--x	ftp
/home/ftp/bin	d--x--x--x	root
/home/ftp/bin/ls	---x--x--x	root
/home/ftp/dev	d--x--x--x	root
/home/ftp/dev/tcp	crw-rw-rw-	root
/home/ftp/dev/zero	crw-rw-rw-	root
/home/ftp/etc	d--x--x--x	root

Table 11-2. Directory and File Requirements for ftp (Cont.)

File or Directory	Permissions	Owner
/home/ftp/etc/group	-r--r--r--	root
/home/ftp/etc/netconfig	-r--r--r--	root
/home/ftp/etc/passwd	-r--r--r--	root
/home/ftp/pub	drwxrwxrwx	ftp
/home/ftp/usr	d--x--x--x	root
/home/ftp/usr/lib	d--x--x--x	root
/home/ftp/usr/lib/libc.so.1	-r-xr-xr-x	root

- The **`/home/ftp/bin/ls`** file should be a copy of **`/bin/ls`**.
- The **`/home/ftp/etc/group`** file should be a copy of **`/etc/group`**.
- The **`/home/ftp/etc/netconfig`** file should be a copy of **`etc/netconfig`**.

3. The **`/home/ftp/lib/libc.so.1`** file should be a copy of **`/lib/libc.so.1`**.

   The **`passwd`** file above should only have entries for the **`ftp`** user and `root`.

This will hide the names of real users from anyone using anonymous **`ftp`**. If a file is owned by a user not listed in the **`passwd`** file, **`ls`** will display the uid instead of the name. The `tcp` and `zero` device nodes should be made with the **`mknod(1M)`** command using the same values that are in **`/dev/tcp`** and **`/dev/zero`**.

A session using anonymous ftp might look like this:

```
venus$ ftp berg
Connected to berg.
220 berg FTP server ready.
Name (berg:stein): anonymous
331 Guest login ok, send ident as password.
Password:  (Here you type some identification string.)
User anonymous logged in.
ftp>
```

Notice the request for `send ident as password.` This is saying that there is no specific password, but that you should send some sort of identification as a password (your name, for example). After connection has been established, you are then given the **`ftp`** prompt that tells you that **`ftp`** is ready to accept transfer commands.

Getting a Listing of Files on the Remote Machine

Once you are connected to a remote **`ftp`** daemon, you can get a listing of the files on the remote machine by using the command **`ls.`** All of the files accessible to you in that directory will then be listed. It is possible to move from one directory to another on the remote machine by means of the **`cd`** command, but it should be noted that unless you have access to those files, you will not be able to transfer them.

Copying Files with ftp Using get and put

The two commands that are used most with **`ftp`** are **`get`** and **`put.`** These commands get a copy of a file from the remote machine, or put a copy onto the remote machine, respectively. To use either command, enter the command followed by *filename*, which is the file to be copied. The **`ftp`** program will report that the transfer has begun, and then when the transfer is complete, along with diagnostic data on how long the transfer took.

```
ftp> get lab1.results
200 PORT command successful.
150 ASCII data connection for lab1.results (129.144.60.88,1163).
226 ASCII Transfer complete.
local: lab1.results remote: lab1.results
1162 bytes received in 0.08 seconds (14 Kbytes/s)
ftp>
ftp> put lab5.data
200 PORT command successful.
150 ASCII data connection for lab5.data (129.144.60.88,1165).
226 Transfer complete.
local: lab5.data remote: lab5.data
1162 bytes sent in 0.04 seconds (28 Kbytes/s)
ftp>
```

Copying Multiple Files Using mget and mput

You can "get" and "put" more than one file at a time. This is done by using the commands **`mget`** and **`mput`**, along with using metacharacters (for example, `*` and `?`)**.** The metacharacter `*` will match anything, while `?` will match any one character. As with **`get`** and **`put`**, **`ftp`** will report when transfer begins. Before each file is transferred, you are asked whether or not you want to transfer it. At this point you answer `y` and the file is transferred, or `n` and the file is skipped. After all matching files have been transferred, you are given the **`ftp`** prompt again.

```
ftp> mput lab*
mput lab1.results? y
200 PORT command successful.
150 ASCII data connection for lab1.results (129.144.60.88,1180).
226 Transfer complete.
local: lab1.results remote: lab1.results
31 bytes sent in 0.02 seconds (1.5 Kbytes/s)
mput lab2.data? n
mput lab3.results? y
200 PORT command successful.
150 ASCII data connection for lab3.results (129.144.60.88,1181).
226 Transfer complete.
local: lab3.results remote: lab3.results
75 bytes sent in 1e-06 seconds (7.3e+04 Kbytes/s)
ftp>
ftp> mget report?.final
mget report1.final? y
200 PORT command successful.
150 ASCII data connection for report1.final (129.144.60.88,1195).
226 ASCII Transfer complete.
local: report1.final remote: report1.final
2605 bytes received in .44 seconds (5.8 Kbytes/s)
mget report2.final? n
ftp>
```

Quitting an ftp Session

When you are finished with **`ftp,`** you can quit by entering the command **`quit`** at the prompt. The connection to the remote daemon will be dropped, and you will be returned to your local shell.

Aborting ftp While Transferring a File

If you are transferring files to or from a remote machine and it goes down, you need to abort the transfer. To abort **`ftp`** when transferring a file, press the interrupt key—usually <BREAK>. You are notified that the transfer was aborted, and then given an **`ftp`** prompt again.

What Happens If There Is No Daemon Present?

Sometimes there is no **`ftp`** daemon running on the remote machine. This can happen if the daemon dies for any reason, or the machine never started one in the first place. If there is no daemon, you can still use **`ftp,`** but the session is non-interactive. When this situation arises, **`ftp`** behaves like **`tftp`** (see the following section).

Transferring Files Non-Interactively Using tftp

The **`tftp`** program is very much like **`ftp`**, except that it is not an interactive process. This means that **`tftp`** does not require that you connect to the remote machine. Rather, when you begin a **`tftp`** session, you are able to issue commands that directly copy files to and from the remote machine, as long as you have an entry in the password database on the remote machine. However, since connection is not maintained between file transfers, you cannot get any directory information from the remote machine.

To use **`tftp`**, enter the command

```
tftp
```

When you see the prompt **`tftp>`**, you are ready to begin transferring files.

Copying Files with tftp Files Using get and put

In order to copy a file from the remote machine, you must use the command **`get`**. The syntax is

`get` *machinename*`:`*file file2 . . . fileN*

where *machinename* is the machine you want to get files from, and *file* is the name of the file you want to get. More than one file can be placed on the command line, with spaces separating the file names.

In order to copy a file to the remote machine, you must use the command **`put`**. The syntax for **`put`** is

`put` *machinename*`:`*file file2 . . . fileN* `[`*remote_directory*`]`

where *machinename* is the machine you want to transfer files to, *file* is the file or a space-separated list of files that you want to transfer, and *remote_directory* is an optional argument showing the specific directory on the remote machine into which you want the files to be placed.

Quitting a tftp Session

When you are finished with **`ftp`**, you can quit by entering the command **`quit`** at the prompt.

Displaying User Information with finger

The **`finger`** command displays information about any user you specify. **`finger`** does not give you information about other machines. It tells you only about other users. In fact,

finger is so user-oriented that it accepts people's real names, as well as their usernames, as arguments.

This is what **finger** tells you:

- the user's login name
- his or her real name
- his or her home directory and login shell
- the last time he or she logged in to the machine from which you are issuing the command
- the last time he or she received mail, and the last time he or she read it
- the name of his or her terminal(s), and how long it's been idle.

Here is a slightly simplified example of two typical **finger** requests. Your output may vary somewhat.

```
venus$ finger moby@sea
[sea]
Login name: moby        In real life: Ishmael Wong
Directory: /home/shipwreck/moby        Shell: /usr/bin/sh
On since Nov 14 06:33:41 on console   4 days 14 hours Idle Time
New mail received Wed Nov 18 20:34:02 1987;
  unread since Wed Nov 18 16:20:24 1987
venus$ finger Henry Stamper
Login name: hank       In real life: Henry Stamper, Jr
Directory: /home/oregon/hank         Shell: /usr/bin/sh
Last login Wed Oct 21 16:16 on ttyp0 from cairo
No unread mail
```

The **finger** command is useful to make sure that the user you are looking for is still active.

Determining If a Machine Is Alive on the Network Using ping

From time to time you will find that a remote machine is not answering your requests. This may indicate network-wide problems or simply that the host is down or disconnected from the network. The **ping** command offers the simplest way to find out if a host on your network is down. Its basic syntax is:

`/usr/sbin/ping` *host* `[` *timeout* `]`

where *host* is the name of the machine in question. The optional *timeout* argument indicates the time in seconds for **ping** to keep trying to reach the machine—20 seconds by default. The **ping(1)** manual page describes additional details.

If you type, for instance,

```
ping elvis
```

you will receive the following message, if host `elvis` is up:

```
elvis is alive
```

indicating that `elvis` has responded to your **`ping.`** If, however, host `elvis` is down or disconnected from the network, you will receive the following message after the specified (or default) timeout has elapsed:

```
no answer from elvis
```

12
Communication Tutorial

12
Communication Tutorial

Introduction

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.

This chapter contains two sections. The first section describes the Basic Networking Utilities (BNU) software that is the basic communications and networking package that comes with UNIX System V and is available to all UNIX users. BNU commands allow users to send and receive files and directories as well as to access networks.

Section two discusses the REXEC software that also is part of the basic UNIX communications and networking package. REXEC lets you log on to a remote machine, execute a command that resides on a remote machine, and run an application or service installed on the remote machine.

To help you take advantage of these capabilities, this chapter will teach you how to use the following commands:

```
ct
cu
rexec
rquery
rx
rl
uucp
uuname
uupick
uustat
uuto
uux
```

Basic Networking Utilities

Basic Networking Utilities (BNU) provides you with a set of commands that allows you to transfer files to a remote machine on a network, check the status of a file transfer, and retrieve files sent by a remote system to a public directory. The commands that send a file on its way to a remote system, or to a specific user on the remote system, are the **uucp** and

uuto commands. The command that monitors and traces the progress of your file transfers is the **uustat** command. The command that lets you collect a file from a public directory, once you have been notified of its arrival, is the **uupick** command.

In addition to file transfer commands, BNU also provides you with a set of commands that allows you to use other machines on the network while logged in to your local system. 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. The **uux** command lets you run commands on a remote system without being logged in to it.

This section tells you how to use all the BNU commands, beginning with **uucp.** In all the examples in this chapter, it is assumed that the machines can communicate with each other.

Transferring a File: The uucp Command

Keeping the overall process in mind, let's begin with the first step - the command to send a file to a remote system. The command **uucp** (short for UNIX-to-UNIX system copy) allows you to copy files between computers. It is not an interactive command. It performs its work silently, invisible to the user. Once you issue this command, you may run other processes.

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.

Syntax for the uucp Command

uucp allows you to send:

- one file to a file or a directory
- multiple files to a directory.

To deliver your file(s), **uucp** must know the full pathname of both the *sourcefile* and the *destinationfile*. However, this does not mean you must type out the full pathname of both files every time you use the **uucp** command. You can use several abbreviations once you become familiar with their formats; **uucp** will expand them to full pathnames.

To specify your *sourcefile* and *destinationfile*, begin by identifying the location of your *sourcefile*, relative to your own current location in the file system. If the *sourcefile* is in your current directory, you can specify it by its name alone (without a path). If the *sourcefile* is not in your current directory, you must specify its full or relative pathname.

How do you specify the *destinationfile*? Because it is on a remote system, the *destinationfile* must always be specified with a pathname that begins with the name of the remote system. After that, however, **uucp** gives you a choice of formats:

- *systemname*!*fullpathname*
- *systemname*!~/[*pathname*]

Here, *fullpathname* is an explicit pathname. ~/ translates to **`/var/spool/uucppublic/`**, **`uucp`**'s public directory on the remote system. *pathname* is a sub-directory, typically having the same name as the recipient's user id.

By default, the only directory to which you can write files is **`/var/spool/uucppublic`**. To write to directories belonging to another user, you must receive write permission from that user or from the administrator.

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 *sourcefiles* or *destinationfiles*. The important distinction in choosing one of these formats is not whether a file is a *sourcefile* or a *destinationfile*, but where you are currently located in the file system relative to the files you are specifying.

For example, let's say you are login `kol` on a system called `mickey`. Your home directory is **`/home/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 **`/home/kol/text`**, so you can specify the *sourcefile* with its relative pathname, **`chap1`**. You can specify the *destinationfile* like this:

- Specify the *destinationfile* with its full pathname:

  ```
  uucp chap1 minnie!/home/wsm/receive/chap1
  ```

  Specify the *destinationfile* with `~/`*pathname*. This expands to the recipient's subdirectory in the public directory on the remote system.

  ```
  uucp chap1 minnie!~/wsm/chap1
  ```

  (The file will go to **`minnie!/var/spool/uucppublic/wsm/chap1`**)

NOTE

The same results can be obtained by omitting **`chap1`** at the end of the previous command line.

Using the uucp Command: Example

Suppose you want to send a file called **`minutes`** to a remote computer named `eagle`. Enter the following command line:

```
$ uucp -m -j minutes eagle!/home/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 **`/home/gws`** in a file named **`minutes`**. When the transfer is complete, you, the sender, are notified by **`mail`**.

The **`-m`** option ensures that you, the sender, are notified by **`mail`** as to whether or not the transfer has succeeded. 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 It 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 *destinationfile*. There are two other ways to specify *destinationfile*:

- The login directory of `gws` can be specified through use of the ~ (tilde), as shown below:

  ```
  eagle!~gws/minutes
  ```

  This is interpreted as:

  ```
  eagle!/home/gws/minutes
  ```

- The `uucppublic` area is referenced by a similar use of the tilde prefix to the pathname. For example:

  ```
  eagle!~/gws/minutes
  ```

  This is interpreted as:

  ```
  /var/spool/uucppublic/gws/minutes
  ```

How uucp Works

This section is an overview of what happens automatically when you issue the **uucp** command. An understanding of the processes involved may help you be aware of the limitations and requirements of the command. For further details, see the *System Administration Manual* and the online *Command Reference.*

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 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 the case, 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. Twice an hour (this is a default, other intervals can be specified), `cron` starts **uudemon.hour**. **uudemon.hour**, in turn, then 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 preceding paragraphs.

The transfer process described generally applies to an active computer (one with calling hardware and networking software). An active computer 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 (**/etc/uucp/Poll**) contains a list of computers that are to be polled in this manner.

Table 12-1 summarizes the syntax and capabilities of the **uucp** command.

Table 12-1. Summary of the uucp Command

Command Recap **uucp** - copies a file from one computer to another		
command	*options*	*arguments*
uucp	**-j**, **-m**, **-s** and others[1]	*sourcefile,destinationfile*
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 **/var/spool/uucppublic**. To write to directories belonging to another user, you must receive write permission from that user and from the administrator. Although there are several ways of representing pathnames as arguments, we recommend that you type full pathnames to avoid confusion.	

1. See the **uucp(1C)** entry in the online *Command Reference* for all available options and an explanation of their capabilities.

Sending Files to the Public Directory: The uuto Command

The **`uuto`** command is a simplified interface to **`uucp`**. It allows you to more easily send files to the public directory (**`/var/spool/uucppublic/`**) of a remote system.

Syntax for the uuto Command

The basic format for the **`uuto`** command is:

uuto *filename(s) 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. It should be noted that **`uuto`** can also route files through intermediate systems on route to the final destination system, providing the intermediate systems permit it. For example, *system!login* can be expressed as *system1!system2!...!login*

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>

Using the uuto Command: Example

Let's take an example and see how this works.

The process of sending a file by **`uuto`** is called 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 number of the job 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 can use the **`uupick`** command (discussed later in this chapter) to retrieve the file.

For the following discussions, assume the information in Table 12-2 is valid:

Table 12-2. Command Example

wombat	your login name
sys1	the name of your local system
marie	the recipient's login name
sys2	the name of the remote system
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, the sender, know when the job has arrived? 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, the sender, when the job has reached the remote system. It is your formal notification that you have indeed successfully transferred the file to the remote system. The message may look something like this:

```
$ mail <RETURN>
>From uucp Fri Feb 3 11:53 EST 1992 remote from sys1
REQUEST: sys1!wombat/money --> sys2!~/receive/marie/sys1/ (marie)
(SYSTEM sys2) copy succeeded
?
```

Table 12-3 is a summary of the syntax and capabilities of the **`uuto`** command.

Table 12-3. Summary of the uuto Command

Command Recap **`uuto`** - sends files to another login		
command	*options*	*arguments*
`uuto`	**`-m`** and others[1]	*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:	You must have read permission for the file(s) you want to send; the file's parent directory must have read and execute permissions for others. The **`-m`** option notifies the sender by **`mail`** when the file has arrived at its destination.	

1. See the **`uuto(1C)`** entry in the online *Command Reference* for all available options and an explanation of their capabilities.

Checking a Job's Status: The uustat Command

Now that you have sent the file, you can go to the next step -- checking the job status. If you would like to determine whether 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 their status.

Using the uustat Command: Example

For example:

```
$ uustat <RETURN>
sys1N2f01 02/03-16:06 S sys2 wombat 10   money
$
```

The elements of the line of this sample status message are as follows:

- `sys1N2f01` is the job number assigned to the job by your host machine.
- `02/03-16:06` is the date and time the job was queued.
- `S` says that this request is to send a file (R means to receive a file).
- `sys2` is the destination machine where the file will be transferred.
- `wombat` is the login name of the person requesting the job.
- `10` is the number of bytes in the file to be transferred.
- `money` is the file to be transferred.

Other status messages and options for the **uustat** command are described in the online *Command Reference.*

That is all there is to sending files and checking the progress of the job. A summary of the syntax and capabilities of the **uustat** command appears in Table 12-4.

Table 12-4. Summary of the uustat Command

Command Recap **uustat** - checks job status of a uucp or uuto job		
command	*options*	*arguments*
uustat	**-k** and others[1]	none
Description:	**uustat** reports the status of all **uucp** and **uuto** jobs you have requested.	
Remarks:	The **-k** option, followed by a job number, allows you to cancel the specified job.	

1. See the **uustat(1C)** entry in the online *Command Reference* for all available options and an explanation of their capabilities.

Retrieving a File: The uupick Command

Now that you know how to send a file and check the progress of the job, let's continue the process from the viewpoint of the user who will be receiving a file. When a file sent by **uuto** reaches the public directory on your UNIX system, you receive a **mail** message. This is your formal notification that you have received a file.

Using the uupick Command: Example

To continue the previous example, the owner of login marie receives the following **mail** message when the file **money** has arrived in the public directory of her system:

```
$ mail <RETURN>
>From uucp Fri Feb  3 16:05 EST 1992 remote from sys2
/var/spool/uucppublic/receive/wombat/sys1/money from sys1!wombat arrived
$
```

The message contains the following information:

- The first line tells you, the receiver, when the file arrived at its destination.
- The first portion of the second line (up to the word "money") gives the pathname of the public directory where the file **money** has been stored.
- The rest of the line (after the word "from") gives the name of the remote system, the remote sender (user), and a status of the file transfer ("arrived").

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, if 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 responses; here are 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 current 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 other files are waiting to be moved, the next one is displayed, followed by the question mark. If not, **`uupick`** exits and the system 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 online *Command Reference.*

You now know how to send a file to a remote system, monitor the progress of the job, and retrieve the file from the public directory. Table 12-5 summarizes the syntax and capabilities of the **`uupick`** command.

Connecting a Remote Terminal: The ct Command

The **`ct`** command connects your computer to a remote terminal equipped with a modem, and allows a user at that terminal to log in. To do this, the command dials the telephone number of the modem. The modem must be able to answer the call. When **`ct`** detects that the call has been answered, it issues a login prompt.

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.

Table 12-5. Summary of the uupick Command

Command Recap **`uupick`** - searches for files sent by uuto or uucp		
command	*options*	*arguments*
`uupick`	**`-ssystem`**	
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. **`uupick`** invoked with the **`-ssystem`** option will search the public directory for files sent only from *system*.	
Remarks:	The question mark (`?`) at the end of the message shows that a response is expected. A complete list of responses appears in the online *Command Reference.*	

Syntax for the ct Command

To execute the **`ct`** command, use this format:

`ct [`*options***`]`** *telno* **<**RETURN>

The argument *telno* is the telephone number of the remote terminal.

Examples for Using the ct Command

Suppose you are logged in on a computer through a local terminal and you want to connect a remote terminal to your computer. Assuming you need to dial a `9` to get an outside telephone line, and the telephone number of the modem on the remote terminal is 555-3497, enter this command line:

```
ct -h -w5 -s1200 9=5553497 <RETURN>
```

NOTE

The equal sign (`=`) represents a secondary dial tone.

`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=5553497 <RETURN>
```

Because you did not specify the **-w** option, if no device is available, **ct** will send you the following message:

```
1 busy dialer at 1200 baud Wait for dialer?
```

If you type `n` (no), the **ct** command will exit. If you type `y` (yes), **ct** will prompt you to specify how long it 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, and does not attempt to call you back.

Table 12-6 summarizes the syntax and capabilities of the **ct** command.

Table 12-6. Summary of the ct Command

Command Recap **ct** - connect computer to remote terminal		
command	*options*	*arguments*
ct	**-h**, **-w**, **-s** and others[1]	*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.	

1. See the **ct(1C)** entry in the online *Command Reference* for all available options and an explanation of their capabilities.

Calling Another UNIX System: The cu Command

The **cu** command connects a remote computer to your computer and allows you to be logged in on both computers simultaneously. This means that you can move back and forth between the two computers, transferring files and executing commands on both, without dropping the connection.

The method used by the **cu** command depends on the information you specify on the command line. You must specify the telephone number or system name of the remote computer. If you specify a telephone 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. You can check for loss of data by using the **sum** command. Before transferring *file* from your local system, issue the **sum** command, using *file* as an argument. Repeat the command on the remote system when *file* is received. The resultant outputs should match, to indicate accurate file transmission.

Syntax for the cu Command

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 (-) represent 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, use 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, with the exception of tilde (~) commands. Table 12-7 and Table 12-8 show the commands you can execute while connected to a remote computer through **cu**.

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.

Table 12-7. Command Strings Used with the cu Command

String	Interpretation
~.	Terminate the link.
~!	Escape to the local computer without dropping the link. To return to the remote computer, type <^d> (control-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 the line ~**...** to the remote computer.
~%break	Transmit a <BREAK> to the remote computer (can also be specified as **~%b**).
~%ifc	Toggles the input flow control setting. When enabled, incoming data may be flow controlled by the local terminal (can also be specified as **~%nostop**).
~%ofc	Toggles the output flow control setting. When enabled, outgoing data may be flow controlled by the remote host (can also be specified as **~%noostop**).
~%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).

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.

Using the cu Command: Example

Suppose you want to connect your computer to a remote computer called eagle. Assuming you need to dial a 9 to get an outside telephone line, and the telephone number for eagle is 555-7867, enter the following command line:

```
cu -s2400 9=5557867 <RETURN>
```

If you do not need to access an outside telephone line, the `9=` characters in the command line are unnecessary. The **`-s2400`** option causes **`cu`** to use a 2400 baud dialer to call `eagle`. If the **`-s`** option is not specified, **`cu`** uses a dialer at the default speed, 1200 baud.

When `eagle` answers the call, **`cu`** notifies you that the connection has been made, and passes `eagle`'s login prompt to you:

```
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. If 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`**.

Table 12-8 summarizes the syntax and capabilities of the **`cu`** command.

Table 12-8. Summary of the cu Command

Command Recap **`cu`** - connects computer to remote computer		
command	*options*	*arguments*
`cu`	**`-s`** and others[1]	*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.	

1. See the **`cu(1C)`** entry in the online *Command Reference* for all available options and an explanation of their capabilities.

Working on a Remote System: The uux Command

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. You will be notified by **mail** if the command you have requested is not allowed to execute (restricted).

Syntax for the uux Command

To execute the **uux** command, follow this format:

uux [*options***]** *commandstring* <RETURN>

The *commandstring* is made up of one or more arguments. All shell special 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 *systemname!* prefix. All arguments that do not contain a *systemname* are interpreted 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).

Using the uux Command: Example

If your computer is hardwired 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`. The **-p** option tells the process to use standard output from **pr minutes** as input to **lp** on `host`.

See the **uux(1C)** manual page in the online *Command Reference* for details. Table 12-9 summarizes the syntax and capabilities of the **uux** command.

The Remote Execution Facility: REXEC

REXEC is a service-based remote execution facility that allows users to execute services on a remote machine. REXEC allows you to do such things as

- log in to a remote machine
- execute a command that resides on a remote machine
- run an application or service installed on the remote machine.

Once you enter an REXEC command and access the remote machine, the process you're running appears to you just as it would if it were running on your local system.

Table 12-9. Summary of the uux Command

Command Recap **uux** - executes commands on a remote computer		
command	*options*	*arguments*
uux	**-p**, and others[1]	*commandstring*
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, the **uux** command can only run the **mail** command. Check with your system administrator to find out if other commands are executable via **uux**.	

1. See the **uux(1C)** entry in the online *Command Reference* for all available options and an explanation of their capabilities.

The services available to you through REXEC are determined by the administrator of the remote machine. Administrators set up and maintain a database of services that can be executed remotely.

Generally, three standard services will be available to you through REXEC. These standard services are defined on the remote machine by default when the REXEC software is installed. The services are

`rx`	which allows you to execute a command or shell script that resides on the remote machine.
`rl`	which allows you to log in to the remote machine from your local machine, provided you have a login on the remote machine and an entry in the remote machine's **/etc/passwd** file.
`rquery`	which allows you to display a list of services that are available to you on the remote machine.

NOTE

Although these standard services are defined by default when REXEC is installed on the remote machine, the remote machine's administrator can choose to remove them from the database of available services. Most likely, however, the `rquery` service will always be defined so that you can check the availability of the other standard services.

In addition to the standard services, the administrator of a remote machine may define other services and make them available to you.

The following section tells you how to use the REXEC facility to execute a process on the remote machine. It then provides you with instructions for using the standard REXEC services.

Using REXEC Services

The user interface to REXEC is the **`rexec`** command. By specifying a remote machine name and a service name as arguments to the **`rexec`** command, a user can execute a process on the remote machine from his or her local system. When the process executes, it appears to the user as if it is running locally. To the remote application or process, the user appears to be on the remote machine.

The syntax of the **`rexec`** command depends on the way your local system is administered. The administrator of the local machine has the option to create a link from an REXEC service to the **`rexec`** command. If the service you want to execute is linked to the **`rexec`** command, you can use an abbreviated command syntax to execute the service.

If no links have been created, the **`rexec`** command has the syntax

> **`rexec`** *host service* **[***parameters***]**

where *host* is the name of the remote machine on which the service resides, *service* is the service name, and *parameters* is the command, including arguments, that you want to run on the remote machine. For example, if you want to see who is logged in to the local system, you enter the **`who`** command. If you want to determine who is logged in to the remote system `strider,` you enter the **`rexec`** command, as follows:

```
rexec strider rx who
```

If a link between the service and the **`rexec`** command has been created, you can enter an abbreviated command with the following syntax:

> *service host* **[***parameters***]**

Again, *service* is the name of the REXEC service you want to run, *host* is the name of the remote machine, and *parameters* is the command, including arguments, that you want to run on the remote machine. If you want to run the **`who`** command on the remote machine `strider,` for example, and `rx` is linked to the **`rexec`** command, you could enter

```
rx strider who
```

The standard services—those defined on the remote machine by default—are linked automatically to the **`rexec`** command. Therefore, you'll most likely use the abbreviated syntax when using the `rx,` `rl,` and `rquery` services. Remember, however, that the administrator of the local machine has the option to remove the links.

Before you can take advantage of the REXEC facility, you must have logins on both the local and the remote systems. The remote machine's administrator creates your remote login, then maps your login on the local machine to it, using a mechanism called ID Mapping. Your login on the remote system is set up as if it were the login of a local user with an entry in **`/etc/passwd,`** a home directory, a **`.profile,`** and so on. For convenience, your remote login and the user account on the remote system associated with that login are referred to throughout this documentation as the "mapped user."

NOTE

If the REXEC facility is protected by the cr1 authentication scheme, you also need a key associated with your login in the cr1 key database. If you fail to connect to a remote machine when REXEC is protected by the cr1 scheme, see your system administrator.

When you execute a service on a remote machine, the process runs as the mapped user. When you execute a shell script on the remote machine, for example, the script is executed by the shell associated with your login on the remote machine, which may not be the same shell you use on your local system.

A remotely executed process cannot access the file system tree on your local system. The working directory associated with the process is the home directory of the mapped user. All files referenced when the service is invoked are relative to the remote machine. For example, if your local machine is `sfzip` and you execute the command

```
rx strider cat \$HOME/.profile
```

the **`.profile`** of the mapped user is displayed, not your **`.profile`** on `sfzip.`

Access to files and subdirectories is controlled on the server side, using the standard UNIX System V permissions scheme. The administrator of your local system controls access to the REXEC facility itself. If REXEC is installed on the local system but you cannot run the **`rexec`** command, see your local administrator. If you are denied access to specific commands or services on the remote machine, contact the remote machine's administrator.

To some extent, you can control the environment in which you execute a remote process by passing environment variables to the remote machine. Passing environment variables is discussed at the end of this section, following instructions for using the standard REXEC services.

Using the rquery Service

The `rquery` service allows you to display the services on a remote machine that are available to you through the REXEC facility. When you execute the `rquery` service and specify a remote machine name, `rquery` runs a command on the remote machine called `rxlist. rxlist` displays all the services that you can access through REXEC.

The output of `rxlist` does not necessarily include all services in the remote machine's REXEC database; because `rxlist` runs on the remote machine as the mapped user, it lists only those services that you, personally, can execute.

To display the available services, you can enter the **`rexec`** command with the following syntax:

```
rexec host rquery
```

Because `rquery` is linked to the **`rexec`** command, you can also use an abbreviated syntax, as follows:

```
rquery host
```

In both commands, *host* is the name of the remote machine.

Using REXEC: Example

To list the available services on a remote machine named `aslan,` using the abbreviated syntax, you would enter

```
rquery aslan
```

NOTE

When you request a display of services available on a remote machine, you do not specify the **`rxlist`** command explicitly. REXEC translates `rquery` into **`rxlist`** on the specified host.

When the command executes, a two-column table similar to the following is displayed on the local system.

```
rx        Remote execution
rl        Remote login
rquery    List available services
```

The first column contains the names of the services on the server that are available to you for remote execution. The second column contains descriptions of the services, as entered by the administrator of the remote machine when the services were defined.

NOTE

Generally, the `rl` service will be defined such that `utmp` entries are created for mapped users. If `utmp` entries are created, then the local logins of mapped users are included in the listing when the **`who`** command is executed. If `utmp` entries are not created, the mapped users' logins do not appear in the **`who`** command's output.

Column 4 of the `rquery` display contains the service definition. The service definition is the command that has been made available for remote execution, plus any macros that define the arguments that the command accepts when it is executed remotely.

Using the rx Service

The `rx` service allows you to execute commands and shell scripts on a remote machine. The arguments passed to the `rx` service are executed by the mapped user's shell; therefore, they can be shell scripts, executable commands, or UNIX built-in commands (such as **`cd`**).

Syntax of the rx Service

Because `rx` is linked to the **`rexec`** command by default, generally you will use the following syntax

`rx` *host command arguments*

where *host* is the name of the remote machine, *command* is the name of the command on the remote machine that you want to run, and *arguments* is the same arguments you could pass to the command, were you running the command locally.

Using the rx Service: Example

Assume, for example, that you want to run the **`cat`** command on a file in the mapper user's home directory on a remote machine named `dopey`. The filename of the file you want to **`cat`** is **`readme`**. You would enter

```
rx dopey cat readme
```

When the command executes, the **`readme`** file scrolls across your screen, as if you had run the **`cat`** command on a file locally.

Since the `rx` service runs a shell on the remote machine, all special quoting and shell metacharacters are handled correctly. For example, suppose you typed the following:

```
rx strider who | fgrep marcus
```

In this case, the **`who`** command runs on the remote machine `strider` and the **`fgrep`** command runs on the local machine. Suppose, however, that you typed the following:

```
rx strider 'who | fgrep marcus'
```

In this case, the string `who | fgrep marcus` executes on the remote machine `strider`.

Similarly, the command

```
rx strider cat readme > readme2
```

creates the **`readme2`** file in the current working directory on the local machine. The command

```
rx strider 'cat readme > readme2'
```

creates the **`readme2`** file in the mapped user's home directory on the remote machine.

Using the rl Service

`rl` lets you connect to a remote machine and execute the login shell of the mapped user. When you run `rl`, `rl` first executes the remote machine's **`/etc/profile`**. It then executes the **`.profile`** of your login on the remote machine and runs the shell specified in the **`.profile.`**

Syntax for the rl Service

Because `rl` is linked to the **`rexec`** command by default, you'll run `rl` using the syntax

`rl` *host*

where *host* is the name of the remote machine you want to access.

Using the rl Service: Example

If you want to log in to a remote machine named `doc,` for example, you would enter

```
rl doc
```

When you make the connection to the remote machine, you are not prompted for a password. Instead, you effectively change machines and continue as the mapped user on the remote machine. The home directory of the mapped user becomes your current working directory.

Passing Environment Variables

REXEC gives you some control of the environment in which you execute a remote process by allowing you to pass environment variables to the remote machine.

The environment variables `SHELL,` `HOME,` and `LOGNAME` are set for the mapped user by the REXEC facility on the remote machine. The values of these variables are obtained from the **`/etc/passwd`** file on the remote machine.

In addition, the variables `PATH` and `TZ` may be set for the mapped user by the listener process on the remote machine, if they were not previously specified for the listener. `PATH` is set to the value of **`/sbin:/usr/sbin:/etc:/usr/bin.`** `TZ` is set to the time zone of the remote machine. If the listener has a value for `PATH` or `TZ,` or both, the value is inherited. The values of `PATH` and `TZ` can be changed by the system administrator of the remote machine.

Other environment variables can be passed from the local system to the remote system when you enter a remote command. The additional variables are specified by assigning a comma-separated list of the variable names to the `RXPORT` variable.

Syntax for Passing Environment Variables

If the `RXPORT` variable has been exported, set `RXPORT` on one line, using the following syntax:

`RXPORT=`*variable1*`,`*variable2*`, . . .`

Then press the `<RETURN>` key and enter the **`rexec`** command, using the syntax `rexec` *host service [parameters]*

or the abbreviated syntax

service host `[`*parameters*`]`

depending on whether or not the service you want to invoke has been linked to the **`rexec`** command.

Passing Environment Variables: Example

Assume you want to execute **`vi`** on the remote machine `strider,` and you may need to set the `TERM` and `EXINIT` variables on the remote machine. To do this, enter the following:

```
RXPORT=TERM,EXINIT
rx strider vi
```

When `RXPORT` is set and exported in the current shell environment, the values of `TERM` and `EXINIT` on the local system will be passed to the remote machine `strider` when `rx` is executed.

13
Programming with awk

13 Programming with awk

Introduction

This chapter describes a programming language that enables you to handle easily the tasks associated with data processing and information retrieval. With **awk,** you can tabulate survey results stored in a file, print various reports summarizing these results, generate form letters, count the occurrences of a string in a file, or reformat a data file used for one application package so it can be used for another application package.

The name **awk** is an acronym formed from the initials of its developers. The name **awk** denotes both the language and the UNIX system command you use to run an **awk** program.

awk is an easy language to learn. It automatically does many things that in other languages you have to program yourself. As a result, many useful **awk** programs are only one or two lines long. Because **awk** programs are usually smaller than equivalent programs in other languages, and because they are interpreted, not compiled, **awk** is also a good language for prototyping.

The first part of this chapter introduces you to the basics of **awk** and is intended to make it easy for you to start writing and running your own **awk** programs. The rest of the chapter describes the complete language and is somewhat less tutorial. If you are an experienced **awk** user, you will find the skeletal summary of the language at the end of the chapter particularly useful.

You should be familiar with the UNIX system and shell programming to use this chapter. Although you don't need other programming experience, some knowledge of the C programming language is beneficial because many constructs found in **awk** are also found in C.

Basic awk

This section provides enough information for you to write and run some of your own programs. Each topic presented in this section is discussed in more detail in later sections.

Program Structure

The basic operation of **awk** is to scan a set of input lines one after another, searching for lines that match any of a set of patterns or conditions you specify. For each pattern, you can specify an action; this action is performed on each line that matches the pattern. Accordingly, an **awk** program is a sequence of pattern-action statements, as Figure 13-1 shows.

```
Structure:
    pattern     {action }
    pattern     { action }
    . . .
Example:
    $1 == “address” { print $2, $3 }
```

Figure 13-1. awk Program Example

The example in Figure 13-1 is a typical **awk** program, consisting of one pattern-action statement. The program prints the second and third fields of each input line whose first field is `address`. In general, **awk** programs work by matching each line of input against each of the patterns in turn. For each pattern that matches, the associated action (which may involve multiple steps) is executed. Then the next line is read and the matching starts again. This process typically continues until all the input has been read.

Either the pattern or the action in a pattern-action statement may be omitted. If there is no action with a pattern, as in

```
$1 == “name”
```

the matching line is printed. If there is no pattern with an action, as in

```
{ print $1, $2 }
```

the action is performed for every input line. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

Usage

You can run an **awk** program two ways. First, you can enter the command

```
$ awk 'pattern-action statements'  optional list of input files  <RETURN>
```

to execute the pattern-action statements on the set of named input files. For example, you could say

```
$ awk '{ print $1, $2 }' file1 file2 <RETURN>
```

Notice that the pattern-action statements are enclosed in single quotes. This protects characters like `$` from being interpreted by the shell and also allows the program to be longer than one line.

If no files are mentioned on the command line, **awk** reads from the standard input. You can also specify that input comes from the standard input by using the hyphen (`-`) as one of the input files.

For example,

```
$ awk '{ print $3, $4 }' file1 - <RETURN>
```

says to read input first from **`file1`** and then from the standard input.

The arrangement above is convenient when the **`awk`** program is short (a few lines). If the program is long, it is often more convenient to put it into a separate file and use the **`-f`** option to fetch it:

```
$ awk -f program_file  optional list of input files  <RETURN>
```

For example, the following command line says to fetch and execute **`myprogram`** on input from the file **`file1`**:

```
$ awk -f myprogram file1 <RETURN>
```

Fields

Normally, **`awk`** reads its input one line, or record, at a time; a record is, by default, a sequence of characters ending with a newline. Then **`awk`** splits each record into fields, where, by default, a field is a string of non-blank, non-tab characters.

As input for many of the **`awk`** programs in this chapter, we use a file called **`countries`**, which contains information about the ten largest countries in the world. (See Table 13-1) Each record contains the name of a country, its area in thousands of square miles, its population in millions, and the continent on which it is located. (Data are from 1978; the U.S.S.R. has been arbitrarily placed in Asia.) The white space between fields is a tab in the original input; a single blank separates `North` and `South` from `America`.

Table 13-1. The Sample Input File countries

USSR	8650	262	Asia
Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	2968	14	Australia
India	1269	637	Asia
Argentina	1072	26	South America
Sudan	968	19	Africa
Algeria	920	18	Africa

This file is typical of the kind of data **`awk`** is good at processing—a mixture of words and numbers separated into fields by blanks and tabs.

The number of fields in a record is determined by the field separator. Fields are normally separated by sequences of blanks and/or tabs, so that the first record of **`countries`** would have four fields, the second five, and so on. It's possible to set the field separator to just tab, so each line would have four fields, matching the meaning of the data; we'll show how to do this shortly. For the time being, we'll use the default: fields separated by blanks and/or tabs. The first field within a line is called `$1,` the second `$2,` and so forth. The entire record is called `$0.`

Printing

If the pattern in a pattern-action statement is omitted, the action is executed for all input lines. The simplest action is to print each line; you can accomplish this with an **`awk`** program consisting of a single `print` statement

```
{ print }
```

so the command line

```
awk '{ print }' countries
```

prints each line of **`countries,`** copying the file to the standard output. The **`print`** statement can also be used to print parts of a record; for instance, the program

```
{ print $1, $3 }
```

prints the first and third fields of each record. Thus

```
awk '{ print $1, $3 }' countries
```

produces as output the following sequence of lines:

```
USSR 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 26
Sudan 19
Algeria 18
```

When printed, items separated by a comma in the `print` statement are separated by the output field separator which, by default, is a single blank. Each line printed is terminated by the output record separator which, by default, is a newline.

NOTE

In the remainder of this chapter, we only show **awk** programs, without the command line that invokes them. Each complete program can be run either by enclosing it in quotes as the first argument of the **awk** command, or by putting it in a file and invoking **awk** with the **-f** flag, as discussed earlier in the section titled "*Usage.*" For example, if no input is mentioned, the input is assumed to be the file **countries.**

Formatted Printing

For more carefully formatted output, **awk** provides a C-like `printf` statement

`printf` *format, expr1, expr2, . . ., exprn*

which prints the $expr_i$'s according to the specification in the string *format.* For example, the **awk** program

```
{ printf "%10s %6d\n", $1, $3 }
```

prints the first field (`$1`) as a string of 10 characters (right-justified), then a space, then the third field (`$3`) as a decimal number in a six-character field, then a newline (`\n`). With input from the file **countries,** this program prints an aligned table:

```
      USSR    262
    Canada     24
     China    866
       USA    219
    Brazil    116
 Australia     14
     India    637
 Argentina     26
     Sudan     19
   Algeria     18
```

With `printf,` no output separators or newlines are produced automatically; you must create them yourself by using `\n` in the format specification. The section "*The printf Statement*" in this chapter contains a full description of `printf.`

Simple Patterns

You can select specific records for printing or other processing by using simple patterns. **awk** has three kinds of patterns. First, you can use patterns called relational expressions that make comparisons. For example, the operator `==` tests for equality. To print the lines for which the fourth field equals the string `Asia`, we can use the program consisting of the single pattern

```
$4 == "Asia"
```

With the file **countries** as input, this program yields

```
USSR    8650   262   Asia
China   3692   866   Asia
India   1269   637   Asia
```

The complete set of comparisons is `>, >=, <, <=, ==` (equal to) and `!=` (not equal to). These comparisons can be used to test both numbers and strings. For example, suppose we want to print only countries with a population greater than 100 million. The program

```
$3 > 100
```

is all that is needed. It prints all lines in which the third field exceeds 100. (Remember that the third field in the file **countries** is the population in millions.)

Second, you can use patterns called regular expressions that search for specified characters to select records. The simplest form of a regular expression is a string of characters enclosed in slashes:

```
/US/
```

This program prints each line that contains the (adjacent) letters `US` anywhere; with the file **countries** as input, it prints

```
USSR   8650   262   Asia
USA    3615   219   North America
```

We will have a lot more to say about regular expressions later in this chapter.

Third, you can use two special patterns, `BEGIN` and `END,` that match before the first record has been read and after the last record has been processed. This program uses `BEGIN` to print a title:

```
BEGIN   { print “Countries of Asia:“ }
/Asia/ { print “     “, $1 }
```

The output is

```
Countries of Asia:
      USSR
      China
      India
```

Simple Actions

We have already seen the simplest action of an **awk** program: printing each input line. Now let's consider how you can use built-in and user-defined variables and functions for other simple actions in a program.

Built-in Variables

Besides reading the input and splitting it into fields, **awk** counts the number of records read and the number of fields within the current record; you can use these counts in your **awk** programs. The variable `NR` is the number of the current record, and `NF` is the number of fields in the record. So the program

```
{ print NR, NF }
```

prints the number of each line and how many fields it has, while

```
{ print NR, $0 }
```

prints each record preceded by its record number.

User-defined Variables

Besides providing built-in variables like `NF` and `NR`, **awk** lets you define your own variables, which you can use for storing data, doing arithmetic, and the like. To illustrate, consider computing the total population and the average population represented by the data in the file **countries:**

```
{ sum = sum + $3 }
END { print “Total population is”, sum, “million”
      print “Average population of”, NR, “countries is”,
      sum/NR }
```

NOTE

awk initializes `sum` to zero before using it.

The first action accumulates the population from the third field; the second action, which is executed after the last input, prints the sum and average:

```
Total population is 2201 million
Average population of 10 countries is 220.1
```

Functions

Built-in functions of **awk** handle common arithmetic and string operations for you. For example, one of the arithmetic functions computes square roots; a string function substitutes one string for another. **awk** also lets you define your own functions. Functions are described in detail in the section titled *“Actions”* in this chapter.

A Handful of Useful One-liners

Although **awk** can be used to write large programs of some complexity, many programs are not much more complicated than what we've seen so far. Here is a collection of other short programs that you may find useful and instructive. Although these programs are not explained here, new constructs they may contain are discussed later in this chapter.

Print last field of each input line:

```
{ print $NF }
```

Print 10th input line:

```
NR == 10
```

Print last input line:

```
    { line = $0}
END { print line }
```

Print input lines that don't have four fields:

```
NF != 4    { print $0, "does not have 4 fields" }
```

Print input lines with more than four fields:

```
NF > 4
```

Print input lines with last field more than 4:

```
$NF > 4
```

Print total number of input lines:

```
END { print NR }
```

Print total number of fields:

```
    { nf = nf + NF }
END { print nf }
```

Print total number of input characters:

```
    { nc = nc + length($0) }
END { print nc + NR }
```
(Adding `NR` includes in the total the number of newlines.)

Print the total number of lines that contain the string Asia:

```
/Asia/{ nlines++ }
END { print nlines }
```
(`nlines`

`++` has the same effect as `nlines = nlines + 1`.)

Error Messages

If you make an error in your **awk** program, you generally get an error message. For example, trying to run the program

```
$3 < 200 { print ( $1 }
```

generates the error messages

```
awk: syntax error at source line 1
 context is
        $3 < 200 { print ( >>> $1 } <<<
awk: illegal statement at source line 1
         1 extra (
```

Some errors may be detected while your program is running. For example, if you try to divide a number by zero, **awk** stops processing and reports the input record number (NR) and the line number in the program.

Patterns

In a pattern-action statement, the pattern is an expression that selects the records for which the associated action is executed. This section describes the kinds of expressions that may be used as patterns.

BEGIN and END

BEGIN and END are two special patterns that give you a way to control initialization and wrap-up in an **awk** program. BEGIN matches before the first input record is read, so any statements in the action part of a BEGIN are done once, before the **awk** command starts to read its first input record. The pattern END matches the end of the input, after the last record has been processed.

The following **awk** program uses BEGIN to set the field separator to tab () and to put column headings on the output. The field separator is stored in a built-in variable called FS. Although FS can be reset at any time, usually the only sensible place is in a BEGIN section, before any input has been read. The second printf statement of the program which is executed for each input line, formats the output into a table, neatly aligned under the column headings. The END action prints the totals. (Notice that a long line can be continued after a comma.)

```
BEGIN { FS = “\t”
        printf “%10s %6s %5s   %s\n”,
               “COUNTRY”, “AREA”, “POP”, “CONTINENT” }
      { printf “%10s %6d %5d   %s\n”, $1, $2, $3, $4
        area = area + $2; pop = pop + $3 }
END   { printf “\n%10s %6d %5d\n”, “TOTAL”, area, pop }
```

With the file **countries** as input, this program produces

```
  COUNTRY   AREA   POP   CONTINENT
     USSR   8650   262   Asia
   Canada   3852    24   North America
    China   3692   866   Asia
      USA   3615   219   North America
   Brazil   3286   116   South America
Australia   2968    14   Australia
    India   1269   637   Asia
Argentina   1072    26   South America
    Sudan    968    19   Africa
  Algeria    920    18   Africa

    TOTAL  30292  2201
```

Relational Expressions

An **awk** pattern can be any expression involving comparisons between strings of characters or numbers. **awk** has six relational operators, and two regular expression matching operators, ~ (tilde) and `!~`, which are discussed in the next section, for making comparisons. Table 13-2 lists and describes these operators:

Table 13-2. awk Comparison Operators

Operator	Meaning
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than
~	matches
!~	does not match

In a comparison, if both operands are numeric, a numeric comparison is made; otherwise, the operands are compared as strings. (Every value might be either a number or a string; usually **awk** can tell what is intended. The section *"Number or String?"* contains more information about this.) Thus, the pattern `$3>100` selects lines where the third field exceeds 100, and the program

```
$1 >= "S"
```

selects lines that begin with the letters S through Z, namely,

```
USSR    8650      262      Asia
USA     3615       219     North America
Sudan    968       19      Africa
```

In the absence of any other information, **awk** treats fields as strings, so the program

```
$1 == $4
```

compares the first and fourth fields as strings of characters, and with the file **countries** as input, prints the single line for which this test succeeds:

```
Australia   2968   14   Australia
```

If both fields appear to be numbers, the comparisons are done numerically.

Regular Expressions

awk provides more powerful patterns for searching for strings of characters than the comparisons illustrated in the previous section. These patterns are called regular expressions, and are like those in **egrep(1)** and **lex(1).** The simplest regular expression is a string of characters enclosed in slashes, like

```
/Asia/
```

This program prints all input records that contain the substring `Asia`. (If a record contains `Asia` as part of a larger string like `Asian` or `Pan-Asiatic`, it is also printed.) In general, if *re* is a regular expression, then the pattern

/re/

matches any line that contains a substring specified by the regular expression *re*.

To restrict a match to a specific field, you use the matching operators ~ (matches) and `!~` (does not match). The program

```
$4 ~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field matches `Asia`, while the program

```
$4 !~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field does not match `Asia`.

In regular expressions, the symbols

```
\ ^ $ . [] * + ? () |
```

are metacharacters with special meanings like the metacharacters in the UNIX shell. For example, the metacharacters `^` and `$` match the beginning and end, respectively, of a string, and the metacharacter `.` (“dot”) matches any single character. Thus,

```
/^.$/
```

matches all records that contain exactly one character.

A group of characters enclosed in brackets matches any one of the enclosed characters; for example, `/[ABC]/` matches records containing any one of `A, B,` or `C` anywhere. Ranges of letters or digits can be abbreviated within brackets: `/[a-zA-Z]/` matches any single letter.

If the first character after the `[` is a `^`, this complements the class so it matches any character not in the set: `/[^a-zA-Z]/` matches any non-letter. The character `+` means "one or more." Thus, the program

```
$2 !~ /^[0-9]+$/
```

prints all records in which the second field is not a string of one or more digits (`^` for beginning of string, `[0-9]+` for one or more digits, and $ for end of string). Programs of this type are often used for data validation.

Parentheses `()` are used for grouping and the character `|` is used for alternatives. The program

```
/(apple|cherry) (pie|tart)/
```

matches lines containing any one of the four substrings `apple pie, apple tart, cherry pie, or cherry tart.`

To turn off the special meaning of a metacharacter, precede it by a \ (backslash). Thus, the program

```
/b\$/
```

prints all lines containing `b` followed by a dollar sign.

In addition to recognizing metacharacters, **awk** recognizes the C programming language escape sequences within regular expressions and strings in the following table:

Table 13-3. C Programming Language Escape Sequences

\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tab
\ddd	octal value *ddd*
\“	quotation mark
\c	any other character *c* literally

For example, to print all lines containing a tab, use the program

```
/\t/
```

awk interprets any string or variable on the right side of a `~` or `!~` as a regular expression. For example, we could have written the program

```
    $2 !~ /^[0-9]+$/
```

as

```
    BEGIN     { digits = “^[0-9]+$“ }
    $2 !~ digits
```

Suppose you want to search for a string of characters like `^[0-9]+$`. When a literal quoted string like `”^[0-9]+$“` is used as a regular expression, one extra level of backslashes is needed to protect regular expression metacharacters. This is because one level of backslashes is removed when a string is originally parsed. If a backslash is needed in front of a character to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string.

For example, suppose you want to match strings containing `b` followed by a dollar sign. The regular expression for this pattern is `b\$.` If you want to create a string to represent this regular expression, you must add one more backslash: `“b\\$“`. The two regular expressions on each of the following lines are equivalent:

```
    x ~ “b\\$“   x ~ /b\$/
    x ~ “b\$“    x ~ /b$/
    x ~ “b$“     x ~ /b$/
    x ~ “\\t”    tx ~ /\t/
```

The precise form of regular expressions and the substrings they match is shown in Table 13-4. The unary operators `*, +,` and `?` have the highest precedence, then concatenation, and then alternation `|`. All operators are left associative. `r` stands for any regular expression.

Table 13-4. awk Regular Expressions

Expression	Matches
c	any non-metacharacter *c*
\c	character *c* literally
^	beginning of string
$	end of string
.	any character but newline
[*s*]	any character in set *s*
[^s]	any character not in set *s*
*r**	zero or more *r*'s
r+	one or more *r*'s
r?	zero or one *r*
(r)	*r*
r1r2	r_1 then r_2 (concatenation)
r1|r2	r_1 or r_2 (alternation)

Combinations of Patterns

A compound pattern combines simpler patterns with parentheses and the logical operators `||` (or), && (and), and *!* (not). For example, suppose you want to print all countries in Asia with a population of more than 500 million. The following program does this by selecting all lines in which the fourth field is `Asia` and the third field exceeds 500:

```
$4 == “Asia” && $3 > 500
```

The program

```
$4 == “Asia” || $4 == “Africa”
```

selects lines with `Asia` or `Africa` as the fourth field. Another way to write the latter query is to use a regular expression with the alternation operator `|` :

```
$4 ~ /^(Asia|Africa)$/
```

The negation operator `!` has the highest precedence, then `&&,` and finally `||.` The operators && and `||` evaluate their operands from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

A pattern range consists of two patterns separated by a comma, as in

pat$_1$, *pat*$_2$ `{ . . . }`

In this case, the action is performed for each line between an occurrence of *pat*$_1$ and the next occurrence of *pat*$_2$ (inclusive). For example, the pattern

```
/Canada/, /Brazil/
```

matches lines starting with the first line that contains the string Canada up through the next occurrence of the string Brazil:

```
Canada  3852    24      North America
China   3692    866     Asia
USA     3615    219     North America
Brazil  3286    116     South America
```

Similarly, since `FNR` is the number of the current record in the current input file (and `FILENAME` is the name of the current input file), the program

```
FNR == 1, FNR == 5 { print FILENAME, $0 }
```

prints the first five records of each input file with the name of the current input file prepended.

Actions

In a pattern-action statement, the action determines what is to be done with the input records that the pattern selects. Actions frequently are simple printing or assignment statements, but they may also be a combination of one or more statements. This section describes the statements that can make up actions.

Built-in Variables

Table 13-5 lists the built-in variables that **awk** maintains. You have already learned some of these; others appear in this and in later sections.

Table 13-5. awk Built-in Variables

Variable	Meaning	Default
ARGC	number of command-line arguments	-
ARGV	array of command-line arguments	-
FILENAME	name of current input file	-
FNR	record number in current file	-
FS	input field separator	blank&tab
NF	number of fields in current record	-
NR	number of records read so far	-
OFMT	output format for numbers	`%.6g`
OFS	output field separator	blank
ORS	output record separator	newline
RS	input record separator	newline
RSTART	index of first character matched by `match`	-
RLENGTH	length of string matched by `match`	-
SUBSEP	subscript separator	“\034“

Arithmetic

Actions can use conventional arithmetic expressions to compute numeric values. As a simple example, suppose you want to print the population density for each country in the file **`countries`**. Since the second field is the area in thousands of square miles and the third field is the population in millions, the expression `1000 * $3 / $2` gives the population density in people per square mile. The program

```
{ printf “%10s %6.1f\n”, $1, 1000 * $3 / $2 }
```

when applied to the file **`countries,`** prints the name of each country and its population density:

```
     USSR   30.3
   Canada    6.2
    China  234.6
      USA   60.6
   Brazil   35.3
Australia    4.7
    India  502.0
Argentina   24.3
    Sudan   19.6
  Algeria   19.6
```

Arithmetic is done internally in floating point. The arithmetic operators are `+, -, *, /, %` (remainder) and `^` (exponentiation; `**` is a synonym). Arithmetic expressions can be created by applying these operators to constants, variables, field names, array elements, functions, and other expressions, all of which are discussed later. Note that **`awk`** recognizes and produces scientific (exponential) notation: `1e6, 1E6, 10e5,` and `1000000` are numerically equal.

`awk` has assignment statements like those found in the C programming language. The simplest form is the assignment statement

v = *e*

where *v* is a variable or field name, and *e* is an expression. For example, to compute the number of Asian countries and their total population, you could write

```
$4 == “Asia” { pop = pop + $3; n = n + 1 }
END          { print “population of”, n,
                   “Asian countries in millions is”, pop }
```

Applied to **`countries,`** this program produces

```
population of 3 Asian countries in millions is 1765
```

The action associated with the pattern `$4 == “Asia”` contains two assignment statements, one to accumulate population and the other to count countries. The variables are not explicitly initialized, yet everything works properly because **`awk`** initializes each variable with the string value `””` and the numeric value `0.`

The assignments in the previous program can be written more concisely using the operators `+=` and `++:`

```
$4 == “Asia”{ pop += $3; ++n }
```

The operator `+=` is borrowed from the C programming language; therefore,

```
pop += $3
```

has the same effect as

```
pop = pop + $3
```

but the `+=` operator is shorter and runs faster. The same is true of the `++` operator, which adds one to a variable.

The abbreviated assignment operators are `+=, -=, *=, /=, %=,` and `^=.` Their meanings are similar:

v op= e

has the same effect as

v = v op e.

The increment operators are `++` and `--`. As in C, they may be used as prefix (`++x`) or postfix (`x++`) operators. If `x` is 1, then `i=++x` increments `x,` then sets `i` to 2, while `i=x++` sets `i` to 1, then increments `x.` An analogous interpretation applies to prefix and postfix `--.`

Assignment and increment and decrement operators may all be used in arithmetic expressions.

We use default initialization to advantage in the following program, which finds the country with the largest population:

```
maxpop < $3 { maxpop = $3; country = $1 }
END         { print country, maxpop }
```

Note, however, that this program would not be correct if all values of `$3` were negative.

awk provides the built-in arithmetic functions listed in Table 13-6.

Table 13-6. awk Built-in Arithmetic Functions

Function	Value Returned
atan2(*y*,*x*)	arctangent of *y/x* in the range -*p* to *p*
cos(*x*)	cosine of *x*, with *x* in radians
exp(*x*)	exponential function of *x*
int(*x*)	integer part of *x* truncated towards 0
log(*x*)	natural logarithm of *x*
rand()	random number between 0 and 1
sin(*x*)	sine of *x*, with *x* in radians
sqrt(*x*)	square root of *x*
srand(*x*)	*x* is new seed for `rand`

x and *y* are arbitrary expressions. The function `rand` returns a pseudo-random floating point number in the range (0,1), and `srand(x)` can be used to set the seed of the generator. If `srand` has no argument, the seed is derived from the time of day.

Strings and String Functions

A string constant is created by enclosing a sequence of characters inside quotation marks, as in `“abc”` or `“hello, everyone”`. String constants may contain the C programming language escape sequences for special characters listed in the “*Regular Expressions*” section in this chapter.

String expressions are created by concatenating constants, variables, field names, array elements, functions, and other expressions. The program

```
{ print NR “:“ $0 }
```

prints each record preceded by its record number and a colon, with no blanks. The three strings representing the record number, the colon, and the record are concatenated and the resulting string is printed. The concatenation operator has no explicit representation other than juxtaposition.

`awk` provides the built-in string functions listed and described in Table 13-7. In this table, *r* represents a regular expression (either as a string or as `/`*r*`/`), *s* and *t* string expressions, and *n* and *p* integers.

Table 13-7. awk Built-in String Functions

Function	Description
gsub(*r*,*s*)	substitute *s* for *r* globally in current record, return number of substitutions
gsub(*r*,*s*,*t*)	substitute *s* for *r* globally in string *t*, return number of substitutions
index(*s*,*t*)	return position of string *t* in *s*, 0 if not present
length(*s*)	return length of *s*
match(*s*,*r*)	return the position in *s* where *r* occurs, 0 if not present
split(*s*,*a*)	split *s* into array *a* on FS, return number of fields
split(*s*,*a*,*r*)	split *s* into array *a* on *r*, return number of fields
sprintf(*fmt*,*exprlist*)	return *exprlist* formatted according to format string fmt
sub(*r*,*s*)	substitute *s* for first *r* in current record, return number of substitutions
sub(*r*,*s*,*t*)	substitute *s* for first *r* in *t*, return number of substitutions
substr(*s*,*p*)	return substring of *s* starting at position *p*
substr(*s*,*p*,*n*)	return substring of *s* of length *n* starting at position p

The functions `sub` and `gsub` are patterned after the substitute command in the text editor **`ed(1).`** The function `gsub`(*r*,*s*,*t*) replaces successive occurrences of substrings matched by the regular expression *r* with the replacement string *s* in the target string *t*. (As in `ed`, the leftmost match is used, and is made as long as possible.) It returns the number of substitutions made. The function `gsub`(*r*,*s*) is a synonym for `gsub(`*r*,*s*`, $0)`. For example, the program

```
{ gsub(/USA/, “United States”); print }
```

transcribes its input, replacing occurrences of `USA` by `United States`. The `sub` functions are similar, except that they only replace the first matching substring in the target string.

The function `index(`*s,t*`)` returns the leftmost position where the string *t* begins in *s*, or zero if *t* does not occur in *s*. The first character in a string is at position 1. For example,

```
index(“banana”, “an”)
```

returns 2.

The `length` function returns the number of characters in its argument string; thus,

```
{ print length($0), $0 }
```

prints each record, preceded by its length. (`$0` does not include the input record separator.) The program

```
length($1) > max { max = length($1); name = $1 }
END              { print name }
```

when applied to the file **`countries,`** prints the longest country name: `Australia`.

The `match`(*s,r*) function returns the position in string *s* where regular expression *r* occurs, or 0 if it does not occur. This function also sets two built-in variables `RSTART` and `RLENGTH.` `RSTART` is set to the starting position of the match in the string; this is the same value as the returned value. `RLENGTH` is set to the length of the matched string. (If a match does not occur, `RSTART` is 0, and `RLENGTH` is -1.) For example, the following program finds the first occurrence of the letter `i` followed by at most one character followed by the letter `a` in a record:

```
{ if (match($0, /i.?a/))
        print RSTART, RLENGTH, $0 }
```

It produces the following output on the file **`countries`**:

```
17 2 USSR8650    262     Asia
26 3 Canada3852    24      North America
3 3 China3692    866     Asia
24 3 USA3615    219     North America
27 3 Brazil3286    116     South America
8 2 Australia2968    14      Australia
4 2 India1269    637     Asia
7 3 Argentina1072    26      South America
17 3 Sudan968     19      Africa
6 2 Algeria920     18      Africa
```

NOTE

`match` matches the leftmost longest matching string. For example, with the record

```
AsiaaaAsiaaaaan
```

as input, the program

```
   { if (match($0, /a+/)) print RSTART,
RLENGTH, $0 }
```

matches the first string of `a`'s and sets `RSTART` to 4 and `RLENGTH` to 3.

The function `sprintf`*(format, expr1, expr2, . . ., exprn)* returns (without printing) a string containing $expr_1$, $expr_2$, . . ., $expr_n$ formatted according to the `printf` specifications in the string *format*. The section titled *"The printf Statement"* in this chapter contains a complete specification of the format conventions.

The statement

```
x = sprintf("%10s %6d", $1, $2)
```

assigns to `x` the string produced by formatting the values of `$1` and `$2` as a ten-character string and a decimal number in a field of width at least six; `x` may be used in any subsequent computation.

The function `substr`(*s,p,n*) returns the substring of *s* that begins at position *p* and is at most *n* characters long. If `substr`(*s,p*) is used, the substring goes to the end of *s;* that is, it consists of the suffix of *s* beginning at position *p*. For example, we could abbreviate the country names in **`countries`** to their first three characters by invoking the program

```
{ $1 = substr($1, 1, 3); print }
```

on this file to produce

```
USS 8650 262 Asia
Can 3852 24 North America
Chi 3692 866 Asia
USA 3615 219 North America
Bra 3286 116 South America
Aus 2968 14 Australia
Ind 1269 637 Asia
Arg 1072 26 South America
Sud 968 19 Africa
Alg 920 18 Africa
```

Note that setting `$1` in the program forces **`awk`** to recompute `$0` and, therefore, the fields are separated by blanks (the default value of `OFS`), not by tabs.

Strings are stuck together (concatenated) merely by writing them one after another in an expression. For example, when invoked on the file **`countries,`**

```
        { s = s substr($1, 1, 3) " " }
    END { print s }
```

prints

```
    USS Can Chi USA Bra Aus Ind Arg Sud Alg
```

by building `s` up a piece at a time from an initially empty string.

Field Variables

The fields of the current record can be referred to by the field variables `$1, $2, . . ., $NF.` Field variables share all of the properties of other variables—they may be used in arithmetic or string operations, and they may have values assigned to them. So, for example, you can divide the second field of the file **`countries`** by 1000 to convert the area from thousands to millions of square miles:

```
    { $2 /= 1000; print }
```

or assign a new string to a field:

```
    BEGIN                   { FS = OFS = "" }
    $4 == "North America"   { $4 = "NA" }
    $4 == "South America"   { $4 = "SA" }
                            { print }
```

The `BEGIN` action in this program resets the input field separator `FS` and the output field separator `OFS` to a tab. Notice that the `print` in the fourth line of the program prints the value of `$0` after it has been modified by previous assignments.

Fields can be accessed by expressions. For example, `$(NF-1)` is the second to last field of the current record. The parentheses are needed: the value of `$NF-1` is 1 less than the value in the last field.

A field variable referring to a nonexistent field, for example, `$(NF+1),` has as its initial value the empty string. A new field can be created, however, by assigning a value to it. For example, the following program invoked on the file **`countries`** creates a fifth field giving the population density:

```
    BEGIN { FS = OFS = "\t"
           { $5 = 1000 * $3 / $2; print }
```

The number of fields can vary from record to record, but usually the implementation limit is 100 fields per record.

Number or String?

Variables, fields and expressions can have both a numeric value and a string value. They take on numeric or string values according to context. For example, in the context of an arithmetic expression like

```
pop += $3
```

`pop` and `$3` must be treated numerically, so their values will be coerced to numeric type if necessary.

In a string context like

```
print $1 ":" $2
```

`$1` and `$2` must be strings to be concatenated, so they will be coerced if necessary.

In an assignment *v* = *e* or *v op*= *e*, the type of *v* becomes the type of *e.* In an ambiguous context like

```
$1 == $2
```

the type of the comparison depends on whether the fields are numeric or string, and this can only be determined when the program runs; it may well differ from record to record.

In comparisons, if both operands are numeric, the comparison is numeric; otherwise, operands are coerced to strings, and the comparison is made on the string values. All field variables are of type string; in addition, each field that contains only a number is also considered numeric. This determination is done at run time. For example, the comparison `"$1 == $2"` will succeed on any pair of the inputs

```
1    1.0    +1    0.1e+1    10E-1   001
```

but will fail on the inputs

```
(null)    0
(null)    0.0
0a        0
1e50      1.0e50
```

There are two idioms for coercing an expression of one type to the other:

number	concatenate a null string to a *number* to coerce it to type string
string + 0	add zero to a *string* to coerce it to type numeric

Thus, to force a string comparison between two fields, use

```
$1 "" == $2 ""
```

The numeric value of a string is the value of any prefix of the string that looks numeric; thus the value of `12.34x` is 12.34, while the value of `x12.34` is zero. The string value of an arithmetic expression is computed by formatting the string with the output format conversion `OFMT`.

Uninitialized variables have numeric value 0 and string value `""`. Nonexistent fields and fields that are explicitly null have only the string value `""`; they are not numeric.

Control Flow Statements

`awk` provides `if-else, while, do-while,` and `for` statements, and statement grouping with braces, as in the C programming language.

The `if` statement syntax is

> `if` *(expression) statement1* `else` *statement2*

The *expression* acting as the conditional has no restrictions; it can include the relational operators `<, <=, >, >=, ==,` and `!=;` the regular expression matching operators `~` and `!~;` the logical operators `||, &&,` and `!;` juxtaposition for concatenation; and parentheses for grouping.

In the `if` statement, **`awk`** first evaluates the *expression*. If it is non-zero and non-null, *statement*$_1$ is executed; otherwise *statement*$_2$ is executed. The `else` part is optional.

A single statement can always be replaced by a statement list enclosed in braces. The statements in the statement list are terminated by newlines or semicolons.

Rewriting the maximum population program from the *"Arithmetic Functions"* section with an `if` statement results in

```
{    if (maxpop < $3) {
          maxpop = $3
          country = $1
     }
}
END  { print country, maxpop }
```

The `while` statement is exactly that of the C programming language:

> `while` *(expression) statement*

The *expression* is evaluated; if it is non-zero and non-null the *statement* is executed and the *expression* is tested again. The cycle repeats as long as the *expression* is non-zero. For example, to print all input fields one per line,

```
{  i = 1
   while (i <= NF) {
      print $i
      i++
   }
}
```

The `for` statement is like that of the C programming language:

> `for (`*expression*$_1$`;` *expression*`;` *expression*$_2$`)` *statement*

It has the same effect as

```
expression_1
while (expression) {
    statement
    expression_2
}
```

so

```
{ for (i = 1; i <= NF; i++) print $i }
```

does the same job as the `while` example shown above. An alternate version of the `for` statement is described in the next section.

The `do` statement has the form

`do` *statement* `while (`*expression*`)`

The *statement* is executed repeatedly until the value of the *expression* becomes zero. Because the test takes place after the execution of the *statement* (at the bottom of the loop), it is always executed at least once. As a result, the `do` statement is used much less often than `while` or `for`, which test for completion at the top of the loop.

The following example of a `do` statement prints all lines except those between `start` and `stop`.

```
/start/ {
           do {
                getline x
           } while (x !~ /stop/)
        }
        { print }
```

The `break` statement causes an immediate exit from an enclosing `while` or `for;` the `continue` statement causes the next iteration to begin. The `next` statement causes **`awk`** to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The `exit` statement causes the program to behave as if the end of the input had occurred; no more input is read, and the `END` action, if any, is executed. Within the `END` action,

`exit` *expr*

causes the program to return the value of *expr* as its exit status. If there is no *expr*, the exit status is zero.

Arrays

awk provides one-dimensional arrays. Arrays and array elements need not be declared; like variables, they spring into existence by being mentioned. An array subscript may be a number or a string.

As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input line to the NRth element of the array x . In fact, it is possible in principle (though perhaps slow) to read the entire input into an array with the **awk** program

```
     { x[NR] = $0 }
END { ... processing ... }
```

The first action merely records each input line in the array x, indexed by line number; processing is done in the END statement.

Array elements may also be named by nonnumeric values. For example, the following program accumulates the total population of Asia and Africa into the associative array pop. The END action prints the total population of these two continents.

```
/Asia/{ pop["Asia"] += $3 }
/Africa/{ pop["Africa"] += $3 }
END { print "Asian population in millions is", pop["Asia"]
      print "African population in millions is", pop["Africa"]
    }
```

On the file **countries,** this program generates

```
Asian population in millions is 1765
African population in millions is 37
```

In this program if you had used pop[Asia] instead of pop["Asia"] the expression would have used the value of the variable Asia as the subscript, and since the variable is uninitialized, the values would have been accumulated in pop[""] .

Suppose your task is to determine the total area in each continent of the file **countries.** Any expression can be used as a subscript in an array reference. Thus

```
area[$4] += $2
```

uses the string in the fourth field of the current input record to index the array area and, in that entry, accumulates the value of the second field:

```
BEGIN { FS = "\t" }
      { area[$4] += $2 }
END   { for (name in area)
                  print name, area[name] }
```

Invoked on the file **`countries,`** this program produces

```
Africa 1888
South America 4358
North America 7467
Australia 2968
Asia 13611
```

This program uses a form of the `for` statement that iterates over all defined subscripts of an array:

`for (`*i* `in` *array) statement*

executes statement with the variable *i* set in turn to each value of *i* for which *array[i]* has been defined. The loop is executed once for each defined subscript, which is chosen in a random order. Results are unpredictable when *i* or *array* is altered during the loop.

`awk` does not provide multi-dimensional arrays, but it does permit a list of subscripts. They are combined into a single subscript with the values separated by an unlikely string (stored in the variable `SUBSEP`).

For example,

```
for (i = 1; i <= 10; i++)
    for (j = 1; j <= 10; j++)
             arr[i,j] = ...
```

creates an array which behaves like a two-dimensional array; the subscript is the concatenation of *i,* `SUBSEP,` and *j.*

You can determine whether a particular subscript *i* occurs in an array *arr* by testing the condition *i* in *arr,* as in

```
if (“Africa” in area) ...
```

This condition performs the test without the side effect of creating `area[“Africa”]`, which would happen if you used

```
if (area[“Africa”] != ““) ...
```

Note that neither is a test of whether the array `area` contains an element with the value `“Africa”`.

It is also possible to split any string into fields in the elements of an array using the built-in function `split.` The function

```
split(“s1:s2:s3”, a, “:“)
```

splits the string `s1:s2:s3` into three fields, using the separator `:`, and stores `s1` in `a[1]`, `s2` in a[2], and `s3` in `a[3]`. The number of fields found, here three, is returned as the value of `split.` The third argument of `split` is a regular expression to be used as the field separator. If the third argument is missing, `FS` is used as the field separator.

An array element may be deleted with the `delete` statement:

`delete` *arrayname[subscript]*

User-defined Functions

`awk` provides user-defined functions. A function is defined as

```
function name(argument-list) {
     statements
}
```

The definition can occur anywhere a pattern-action statement can. The argument list is a list of variable names separated by commas; within the body of the function these variables refer to the actual parameters when the function is called. No space must be left between the function name and the left parenthesis of the argument list when the function is called; otherwise it looks like a concatenation. For example, the following program defines and tests the usual recursive factorial function (of course, using some input other than the file **`countries`**):

```
function fact(n) {
     if (n <= 1)
          return 1
     else
          return n * fact(n-1)
}
{ print $1 "! is " fact($1) }
```

Array arguments are passed by reference, as in C, so it is possible for the function to alter array elements or create new ones. Scalar arguments are passed by value, however, so the function cannot affect their values outside. Within a function, formal parameters are local variables, but all other variables are global. (You can have any number of extra formal parameters that are used only as local variables.) The `return` statement is optional, but the returned value is undefined if it is not included.

Some Lexical Conventions

Comments may be placed in **`awk`** programs; they begin with the character # and end at the end of the line, as in

```
print x, y     # this is a comment
```

Statements in an **`awk`** program normally occupy a single line. Several statements may occur on a single line if they are separated by semicolons. A long statement may be continued over several lines by terminating each continued line by a backslash. (It is not possible to continue a "…" string.) This explicit continuation is rarely necessary, however, since statements continue automatically if the line ends with a comma. (For example, this might occur in a `print` or printf statement) or after the operators && and `||`.

Several pattern-action statements may appear on a single line if separated by semicolons.

Output

The `print` and `printf` statements are the two primary constructs that generate output. The `print` statement is used to generate simple output; `printf` is used for more carefully formatted output. Like the shell, **`awk`** lets you redirect output, so that output from `print` and `printf` can be directed to files and pipes. This section describes the use of these two statements.

The print Statement

The statement

`print` $expr_1$, $expr_2$, ..., $expr_n$

prints the string value of each expression separated by the output field separator followed by the output record separator. The statement

```
print
```

is an abbreviation for

```
print $0
```

To print an empty line, use

```
print ““
```

Output Separators

The output field separator and record separator are held in the built-in variables `OFS` and `ORS.` Initially, `OFS` is set to a single blank and `ORS` to a single newline, but these values can be changed at any time. For example, the following program prints the first and second fields of each record with a colon between the fields and two newlines after the second field:

```
BEGIN { OFS = “:“; ORS = “\n\n” }
       { print $1, $2 }
```

Notice that

```
    { print $1 $2 }
```

prints the first and second fields with no intervening output field separator because `$1 $2` is a string consisting of the concatenation of the first two fields.

The printf Statement

`awk's` `printf` statement is the same as that in C except that the `*` format specifier is not supported. The `printf` statement has the general form

`printf` *format*, $expr_1$, $expr_2$, . . ., $expr_n$

where *format* is a string that contains both information to be printed and specifications on what conversions are to be performed on the expressions in the argument list, as in Table 13-8. Each specification begins with a `%,` ends with a letter that determines the conversion, and may include:

`-`	Left-justify expression in its field.
width	Pad field to this width as needed; fields that begin with a leading 0 are padded with zeros.
`.prec`	Specify maximum string width or digits to right of decimal point.

Table 13-8 lists the **`printf`** conversion characters.

Table 13-8. awk printf Conversion Characters

Character	Prints Expression as
c	single character
d	decimal number
e	[-]*d.ddddddE[+-]dd*
f	[-]*ddd.dddddd*
g	`e` or `f` conversion, whichever is shorter, with nonsignificant zeros suppressed
o	unsigned octal number
s	string
x	unsigned hexadecimal number
%	print a `%`; no argument is converted

Below are some examples of `printf` statements along with the corresponding output:

```
printf “%d”, 99/249
printf “%e”, 99/24.950000e+01
printf “%f”, 99/249.500000
printf “%6.2f”, 99/249.50
printf “%g”, 99/249.5
printf “%o”, 99143
printf “%06o”, 99000143
printf “%x”, 9963
printf “|%s|“, “January”|January|
printf “|%10s|“, “January”|   January|
printf “|%-10s|“, “January”|January   |
printf “|%.3s|“, “January”|Jan|
printf “|%10.3s|“, “January”|       Jan|
printf “|%-10.3s|“, “January”|Jan       |
printf “%%“%
```

The default output format of numbers is `%.6g;` this can be changed by assigning a new value to `OFMT. OFMT` also controls the conversion of numeric values to strings for concatenation and creation of array subscripts.

Output to Files

You can print output to files instead of to the standard output by using the `>` and `>>` redirection operators. For example, the following program invoked on the file **`countries`** prints all lines where the population (third field) is bigger than 100 into a file called **`bigpop,`** and all other lines into a file called **`smallpop`**:

```
$3 > 100   { print $1, $3 >“bigpop” }
$3 <= 100 { print $1, $3 >“smallpop” }
```

Notice that the filenames have to be quoted; without quotes, `bigpop` and `smallpop` are merely uninitialized variables. If the output filenames were created by an expression, they would also have to be enclosed in parentheses:

```
$4 ~ /North America/ { print $1 > (“tmp” FILENAME) }
```

because the `>` operator has higher precedence than concatenation; without parentheses, the concatenation of `tmp` and `FILENAME` would not work.

NOTE

Files are opened once in an **`awk`** program. If `>` is used to open a file, its original contents are overwritten. But if `>>` is used to open a file, its contents are preserved and the output is appended to the file. Once the file has been opened, the two operators have the same effect.

Output to Pipes

You can also direct printing to a pipe with a command on the other end, instead of to a file. The statement

```
print | “command-line”
```

causes the output of **`print`** to be piped into the *command-line.*

Although they are shown here as literal strings enclosed in quotes, the *command-line* and filenames can come from variables and the return values from functions.

Suppose you want to create a list of continent-population pairs, sorted alphabetically by continent. The **`awk`** program below accumulates the population values in the third field for each of the distinct continent names in the fourth field in an array called `pop`. Then it prints each continent and its population, and pipes this output into the **`sort`** command.

```
BEGIN   { FS = “\t” }
        { pop[$4] += $3 }
END     { for (c in pop)
              print c “:“ pop[c] | “sort” }
```

Invoked on the file **`countries,`** this program yields

```
Africa:37
Asia:1765
Australia:14
North America:243
South America:142
```

In all these `print` statements involving redirection of output, the files or pipes are identified by their names (that is, the pipe above is literally named `sort`), but they are created and opened only once in the entire run. So, in the last example, for all `c` in `pop`, only one sort pipe is open.

There is a limit to the number of files that can be open simultaneously. The statement `close`*(file)* closes a file or pipe; *file* is the string used to create it in the first place, as in

```
close(“sort”)
```

When opening or closing a file, different strings are different commands.

Input

The most common way to give input to an **`awk`** program is to name on the command line the file(s) that contains the input. This is the method used in this chapter; however, several other methods can be used. Each of these is described in this section.

Files and Pipes

You can provide input to an **awk** program by putting the input data into a file, say **awk data,** and then executing

```
$ awk 'program' awkdata <RETURN>
```

If no filenames are given, **awk** reads its standard input; thus, a second common arrangement is to have another program pipe its output into **awk.** For example, **egrep(1)** selects input lines containing a specified regular expression, but it can do so faster than **awk,** since this is the only thing it does. We could, therefore, invoke the pipe

```
$ egrep 'Asia' countries | awk '. . .' <RETURN>
```

egrep quickly finds the lines containing `Asia` and passes them on to the **awk** program for subsequent processing.

Input Separators

With the default setting of the field separator `FS,` input fields are separated by blanks or tabs, and leading blanks are discarded, so each of these lines has the same first field:

```
    field1   field2
  field1
field1
```

When the field separator is a tab, however, leading blanks are not discarded.

The field separator can be set to any regular expression by assigning a value to the built-in variable `FS.` For example,

```
BEGIN { FS = ",[\t]*|([\t]+)" }
```

makes into field separators every string consisting of a comma followed by blanks or tabs and every string of blanks or tabs with no comma. `FS` can also be set on the command line with the **-F** argument:

```
$ awk -F'(,[ \t]*)|([ \t]+)' '. . .' <RETURN>
```

behaves the same as the previous example. Regular expressions used as field separators match the leftmost longest occurrences (as in `sub`), but do not match null strings.

Multi-line Records

Records are normally separated by newlines, so that each line is a record; but this too can be changed, though only in a limited way. If the built-in record separator variable `RS` is set to the empty string, as in

```
BEGIN   { RS = "" }
```

then input records can be several lines long; a sequence of empty lines separates records. A common way to process multiple-line records is to use

```
BEGIN   { RS = ““; FS = “\n” }
```

to set the record separator to an empty line and the field separator to a newline. Each line is then one field. However, the length of a record is limited; it is usually about 2500 characters. *“The getline Function”* and *“Cooperation with the Shell”* sections in this chapter show other examples of processing multi-line records.

The getline Function

`awk`'s facility for automatically breaking its input into records that are more than one line long is not adequate for some tasks. For example, if records are not separated by blank lines, but by something more complicated, merely setting `RS` to null doesn't work. In such cases, the program must manage the splitting of each record into fields. Here are some suggestions.

The function `getline` can be used to read input either from the current input or from a file or pipe, by redirection analogous to **`printf.`** By itself, `getline` fetches the next input record and performs the normal field-splitting operations on it. It sets `NF, NR,` and `FNR. getline` returns 1 if there was a record present, 0 if the end-of-file was encountered, and -1 if some error occurred (such as failure to open a file).

To illustrate, suppose you have input data consisting of multi-line records, each of which begins with a line beginning with `START` and ends with a line beginning with `STOP`. The following **`awk`** program processes these multi-line records, a line at a time, putting the lines of the record into consecutive entries of an array

```
f[1] f[2] ... f[nf]
```

Once the line containing `STOP` is encountered, the record can be processed from the data in the `f` array:

```
/^START/ {
          f[nf=1] = $0
          while (getline && $0 !~ /^STOP/)
               f[++nf] = $0
          # now process the data in f[1]...f[nf]
          ...
}
```

Notice that this code uses the fact that && evaluates its operands left to right and stops as soon as one is true.

The same job can also be done by the following program:

```
/^START/ && nf==0{ f[nf=1] = $0 }
nf > 1{ f[++nf] = $0 }
/^STOP/{ # now process the data in f[1]...f[nf]
      ...
      nf = 0
}
```

The statement

```
getline x
```

reads the next record into the variable `x`. No splitting is done; `NF` is not set. The statement

```
getline <"file"
```

reads from **`file`** instead of the current input. It has no effect on `NR` or `FNR,` but field splitting is performed and `NF` is set.

The statement

```
getline x <"file"
```

gets the next record from **`file`** into `x`; no splitting is done, and `NF, NR` and `FNR` are untouched.

If a filename is an expression, it should be in parentheses for evaluation:

```
while ( getline x < (ARGV[1] ARGV[2]) ) {  ... }
```

because the **<** has precedence over concatenation. Without parentheses, a statement such as

```
getline x < "tmp" FILENAME
```

sets `x` to read the file **`tmp`** and not **`tmp`** *<value of FILENAME>*. Also, if you use this `getline` statement form, a statement like

```
while ( getline x < file ) { ... }
```

loops forever if the file cannot be read because `getline` returns -1, not zero if an error occurs. A better way to write this test is

```
while ( getline x < file > 0) { ... }
```

You can also pipe the output of another command directly into `getline.` For example, the statement

```
while ("who" | getline)
     n++
```

executes `who` and pipes its output into `getline.` Each iteration of the `while` loop reads one more line and increments the variable `n,` so after the `while` loop terminates, `n` contains a count of the number of users. Similarly, the statement

```
“date” | getline d
```

pipes the output of `date` into the variable `d,` thus setting `d` to the current date. Table 13-9 summarizes the `getline` function.

Table 13-9. getline Function

Form	Sets
getline	$0, NF, NR, FNR
getline *var*	*var*, NR, FNR
getline <*file*	$0, NF
getline *var* <*file*	*var*
cmd | getline	$0, NF
cmd | getline *var*	*var*

Command-line Arguments

The command-line arguments are available to an **`awk`** program: the array `ARGV` contains the elements `ARGV[0],...,` `ARGV[ARGC-1];` as in C, `ARGC` is the count. `ARGV[0]` is the name of the program (generally **`awk);`** the remaining arguments are whatever was provided (excluding the program and any optional arguments) when **`awk`** is invoked. The following command line contains an **`awk`** program that echoes the arguments that appear after the program name:

```
awk '
BEGIN {
   for (i = 1; i < ARGC; i++)
         printf “%s “, ARGV[i]
   printf “\n“
}' $*
```

The arguments may be modified or added to; `ARGC` may be altered. As each input file ends, **`awk`** treats the next non-null element of `ARGV` (up to the current value of `ARGC-1)` as the name of the next input file.

One exception to the rule that an argument is a filename is when it is in the form

var=value

Then the variable *var* is set to the value *value*, as if by assignment. Such an argument is not treated like a filename. If *value* is a string, no quotes are needed.

Using awk with Other Commands and the Shell

awk is most powerful when it is used in conjunction with other programs. This section discusses some of the ways in which **awk** programs cooperate with other commands.

The system Function

The built-in function `system`*(command-line)* executes the command *command-line,* which may be a string computed by, for example, the built-in function `sprintf.` The value returned by `system` is the return status of the command executed.

For example, the program

```
$1 == “#include” { gsub(/[<>“]/, ““, $2);
 system(“cat “ $2) }
```

calls the command **cat** to print the file named in the second field of every input record whose first field is `#include,` after stripping any `<, >,` or ″ that might be present.

Cooperation with the Shell

In all the examples thus far, the **awk** program was in a file and was fetched from there using the **-f** flag, or it appeared on the command line enclosed in single quotes, as in

```
awk '{ print $1 }' . . .
```

Since **awk** uses many of the same characters as the shell does, such as `$` and ″, surrounding the **awk** program with single quotes ensures that the shell will pass the entire program unchanged to the **awk** interpreter.

Now, consider writing a command **addr** that will search a file **addresslist** for name, address and telephone information. Suppose that **addresslist** contains names and addresses in which a typical entry is a multi-line record such as

```
G. R. Emlin
600 Mountain Avenue
Murray Hill, NJ 07974
201-555-1234
```

Records are separated by a single blank line.

You want to search the address list by issuing commands like

```
addr Emlin
```

That is easily done by a program of the form

```
awk '
BEGIN{ RS = ““ }
/Emlin/
' addresslist
```

The problem is how to get a different search pattern into the program each time it is run.

There are several ways to do this. One way is to create a file called **`addr`** that contains

```
awk '
BEGIN{ RS = ““ }
/'$1'/
' addresslist
```

The quotes are critical here. The **`awk`** program is only one argument, even though there are two sets of quotes because quotes do not nest. The `$1` is outside the quotes, visible to the shell, which therefore replaces it by the pattern `Emlin` when the command **`addr`** `Emlin` is invoked. On a UNIX system, **`addr`** can be made executable by changing its mode with the following command:

```
chmod +x addr
```

A second way to implement **`addr`** relies on the fact that the shell substitutes for `$` parameters within double quotes:

```
awk “
BEGIN{ RS = \\′ \\′ }
/$1/
“ addresslist
```

Therefore, you must protect the quotes defining `RS` with backslashes, so that the shell passes them on to **`awk`** without interpretation. `$1` is recognized as a parameter, however, so the shell replaces it by the pattern when the command **`addr`** *pattern* is invoked.

A third way to implement **`addr`** is to use `ARGV` to pass the regular expression to an **`awk`** program that explicitly reads through the address list with `getline:`

```
awk '
BEGIN   { RS = ““
          while (getline < “addresslist”)
             if ($0 ~ ARGV[1])
                print $0
} ' $*
```

All processing is done in the `BEGIN` action.

Notice that any regular expression can be passed to **`addr;`** in particular, it is possible to retrieve by parts of an address or telephone number as well as by name.

Example Applications

awk has been used in surprising ways: to implement database systems and a variety of compilers and assemblers, in addition to the more traditional tasks of information retrieval, data manipulation, and report generation. Invariably, the **awk** programs are significantly shorter than equivalent programs written in more conventional programming languages such as Pascal or C. This section presents a few more examples to illustrate some additional **awk** programs.

Generating Reports

awk is especially useful for producing reports that summarize and format information. Suppose you want to produce a report from the file **countries** in which the continents are listed alphabetically, and the countries on each continent are listed after in decreasing order of population:

```
Africa:
    Sudan          19
    Algeria        18

Asia:
    China         866
    India         637
    USSR          262

Australia:
    Australia      14

North America:
    USA           219
    Canada         24

South America:
    Brazil        116
    Argentina      26
```

As with many data processing tasks, it is much easier to produce this report in several stages. First, create a list of continent-country-population triples, in which each field is separated by a colon. This can be done with the following program `triples,` which uses an array `pop` indexed by subscripts of the form *continent:country* to store the population of a given country. The print statement in the `END` section of the program creates the list of continent-country-population triples that are piped to the `sort` routine.

```
BEGIN { FS = “\t” }
      { pop[$4 “:“ $1] += $3 }
END   { for (cc in pop)
          print cc “:“ pop[cc] | “sort -t: +0 -1 +2nr” }
```

The arguments for **sort** deserve special mention. The **-t**: argument tells **sort** to use **:** as its field separator. The `+0 -1` arguments make the first field the primary sort key. In general, *+i* **-j** makes fields *i+1*, *i+2*, . . ., *j t*he sort key. If **-j** is omitted, the fields from *i+1* to the end of the record are used. The `+2nr` argument makes the third field, numerically decreasing, the secondary sort key (`n` is for numeric, `r` for reverse order). Invoked on the file **`countries,`** this program produces as output

```
Africa:Sudan:19
Africa:Algeria:18
Asia:China:866
Asia:India:637
Asia:USSR:262
Australia:Australia:14
North America:USA:219
North America:Canada:24
South America:Brazil:116
South America:Argentina:26
```

This output is in the right order but the wrong format. To transform the output into the desired form, run it through a second **`awk`** program **`format:`**

```
BEGIN  { FS = “:“ }
{      if ($1 != prev) {
            print “\n” $1 “:“
            prev = $1
       }
       printf “\t%-10s %6d\n”, $2, $3
}
```

This is a control-break program that prints only the first occurrence of a continent name and formats the country-population lines associated with that continent in the desired manner. The command line

```
$ awk -f triples countries | awk -f format <RETURN>
```

gives the desired report. As this example suggests, complex data transformation and formatting tasks can often be reduced to a few simple **`awk`** commands and **`sort`**s.

Additional Examples

Word Frequencies

Our first example illustrates associative arrays for counting. Suppose you want to count the number of times each word appears in the input, where a word is any contiguous sequence of non-blank, non-tab characters. The following program prints the word frequencies, sorted in decreasing order.

```
{ for (w = 1; w <= NF; w++) count[$w]++ }
END { for (w in count) print count[w], w | “sort -nr” }
```

The first statement uses the array `count` to accumulate the number of times each word is used. Once the input has been read, the second `for` loop pipes the final count along with each word into the **`sort`** command.

Accumulation

Suppose we have two files, **`deposits`** and **`withdrawals,`** of records containing a name field and an amount field. For each name we want to print the net balance determined by subtracting the total withdrawals from the total deposits for each name. The net balance can be computed by the following program:

```
awk '
FILENAME == “deposits”     { balance[$1] += $2 }
FILENAME == “withdrawals”  { balance[$1] -= $2 }
END                        { for (name in balance)
                                 print name, balance[name]
} ' deposits withdrawals
```

The first statement uses the array `balance` to accumulate the total amount for each name in the file **`deposits.`** The second statement subtracts associated withdrawals from each total. If only withdrawals are associated with a name, an entry for that name is created by the second statement. The `END` action prints each name with its net balance.

Random Choice

The following function prints (in order) `k` random elements from the first `n` elements of the array `A.` In the program, `k` is the number of entries that still need to be printed, and `n` is the number of elements yet to be examined. The decision of whether to print the *ith* element is determined by the test `rand() < k/n.`

```
function choose(A, k, n, i) {
       for (i = 1; n > 0; i++)
             if (rand() < k/n--) {
                    print A[i]
                    k--
             }
       }
}
```

Shell Facility

The following **`awk`** program roughly simulates the history facility of the UNIX system shell. A line containing only `=` re-executes the last command executed. A line beginning with `=` *cmd* re-executes the last command whose invocation included the string *cmd.* Otherwise, the current line is executed.

```
$1 == “=“ { if (NF == 1)
                system(x[NR] = x[NR-1])
            else
                for (i = NR-1; i > 0; i--)
                        if (x[i] ~ $2) {
                                system(x[NR] = x[i])
                                break
                        }
            next }

/./         { system(x[NR] = $0) }
```

Form-letter Generation

The following program generates form letters, using a template stored in a file called **`form.letter`:**

```
This is a form letter.
The first field is $1, the second $2, the third $3.
The third is $3, second is $2, and first is $1.
```

and replacement text of this form:

```
field 1|field 2|field 3
one|two|three
a|b|c
```

The `BEGIN` action stores the template in the array `template`; the remaining action cycles through the input data, using **`gsub`** to replace template fields of the form `$n` with the corresponding data fields.

```
BEGIN {FS = “|“
                while (getline <“form.letter”)
                line[++n] = $0
}
{       for (i = 1; i <= n; i++) {
                s = line[i]
                for (j = 1; j <= NF; j++)
                           gsub(“\\$“j, $j, s)
                print s
        }
}
```

In all such examples, a prudent strategy is to start with a small version and expand it, trying out each aspect before moving on to the next.

awk Summary

Command Line

`awk` *program filenames*
`awk -f` *program-file filenames*
`awk -F``s` *sets field separator to string s;* **`-Ft`** *sets separator to tab*

Patterns

`BEGINEND`
`/`*regular expression/*
relational expression
pattern && pattern
pattern || pattern
(pattern)
!pattern
pattern, pattern

Control Flow Statements

`if` *(expr) statement* `[else` *statement*`]`
`if` *(subscript in array) statement* `[else` *statement*`]`
`while (`*expr*`)` *statement*
`for` *(expr; expr; expr) statement*
`for` *(var in array) statement*
`do` *statement* `while (`*expr*`)`
`break`
`continue`
`next`
`exit [`*expr*`]`
`return [`*expr*`]`

Input-Output

Table 13-10 lists and describes the Input/Output functions:

Table 13-10. Input Output Functions

close(*filename*)	close file
getline	set `$0` from next input record; set `NF, NR, FNR`
getline <*file*	set `$0` from next record of *file*; set `NF`

Table 13-10. Input Output Functions (Cont.)

getline *var*	set *var* from next input record; set `NR, FNR`
getline *var <file*	set *var* from next record of *file*
print	print current record
print *expr-list*	print expressions
print *expr-list >file*	print expressions on *file*
printf *fmt, expr-list*	format and print
printf *fmt, expr-list >file*	format and print on *file*
system(*cmd-line*)	execute command **`cmd-line`**, return status

In `print` and `printf` in Table 13-10, *>>file* appends to the *file*, and `|` *command* writes on a pipe. Similarly, *command* `| getline` pipes into `getline. getline` returns 0 on end of file, and -1 on error.

Functions

`func` *name(parameter list) { statement }*
`function` *name(parameter list) { statement }*
function-name(expr, expr, . . .)

String Functions

Table 13-11 lists and describes the string functions:

Table 13-11. String Functions

`gsub`(*r,s,t*)	substitute string *s* for each substring matching regular expression *r* in string *t*, return number of substitutions; if *t* omitted, use `$0`
`index`(*s,t*)	return index of string *t* in string *s*, or 0 if not present
`length`(*s*)	return length of string *s*
`match`(*s,r*)	return position in *s* where regular expression *r* occurs, or 0 if *r* is not present
`split`(*s,a,r*)	split string *s* into array *a* on regular expression *r*, return number of fields; if *r* omitted, `FS` is used in its place
`sprintf(`*fmt, expr-list)*	print *expr-list* according to *fmt*, return resulting string
`sub`(*r,s,t*)	like `gsub` except only the first matching substring is replaced
`substr`(*s,i,n*)	return *n*-char substring of *s* starting at *i*; if *n* omitted, use rest of *s*

Arithmetic Functions

Table 13-12 lists and describes the arithmetic functions:

Table 13-12. Arithmetic Functions

atan2(*y*,*x*)	arctangent of *y/x* in radians
cos(*expr*)	cosine (angle in radians)
exp(*expr*)	exponential
int(*expr*)	truncate to integer
log(*expr*)	natural logarithm
rand()	random number between 0 and 1
sin(*expr*)	sine (angle in radians)
sqrt(*expr*)	square root
srand(*expr*)	new seed for random number generator; use time of day if no expr

Operators (Increasing Precedence)

Table 13-13 lists and describes the awk operators:

Table 13-13. awk Operators

= += -= *= /= %= ^=	assignment
?:	conditional expression
||	logical OR
&&	logical AND
~ !~	regular expression match, negated match
< <= > >= != ==	relationals
blank	string concatenation
+ -	add, subtract
* / %	multiply, divide, mod
+ - !	unary plus, unary minus, logical negation
^	exponentiation (** is a synonym)
++ --	increment, decrement (prefix and postfix)
$	field

Regular Expressions (Increasing Precedence)

Table 13-14 lists and describes the regular expressions:

Table 13-14. Regular Expressions

c	matches non-metacharacter *c*
\c	matches literal character *c*
.	matches any character but newline
^	matches beginning of line or string
$	matches end of line or string
[*abc...*]	character class matches any of *abc...*
[^*abc...*]	negated class matches any but *abc...* and newline
r1|*r2*	matches either *r1* or *r2*
r1r2	concatenation: matches *r1*, then *r2*
r+	matches one or more *r*'s
*r**	matches zero or more *r*'s
r?	matches zero or one *r*'s
(r)	grouping: matches *r*

Built-in Variables

Table 13-15 lists and describes the built-in variables:

Table 13-15. Built-in Variables

ARGC	number of command-line arguments
ARGV	array of command-line arguments (`0..ARGC-1`)
FILENAME	name of current input file
FNR	input record number in current file
FS	input field separator (default blank)
NF	number of fields in current input record
NR	input record number since beginning
OFMT	output format for numbers (default `%.6g`)
OFS	output field separator (default blank)
ORS	output record separator (default newline)
RS	input record separator (default newline)

Table 13-15. Built-in Variables (Cont.)

RSTART	index of first character matched by `match()`; 0 if no match
RLENGTH	length of string matched by `match()`; -1 if no match
SUBSEP	separates multiple subscripts in array elements; default `\034`

Limits

Any particular implementation of **awk** enforces some limits. Here are typical values:

```
100 fields
2500 characters per input record
2500 characters per output record
1024 characters per individual field
1024 characters per printf string
400 characters maximum quoted string
400 characters in character class
15 open files
1 pipe
numbers are limited to what can be represented on the
local machine, for example, 1e-38..1e+38
```

Initialization, Comparison, and Type Coercion

Each variable and field can potentially be a string or a number or both at any time. When a variable is set by the assignment

var = *expr*

its type is set to that of the expression. (Assignment includes `+=`, `-=`, and so on.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in

```
v1 = v2
```

then the type of `v1` becomes that of `v2`.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by subterfuges such as

```
expr + 0
```

and to string by

```
expr ““
```

(that is, concatenation with a null string).

Uninitialized variables have the numeric value `0` and the string value `""`. Accordingly, if `x` is uninitialized,

```
if (x) ...
```

is false, and

```
if (!x) ...
if (x == 0) ...
if (x == "") ...
```

are all true. But the following is false:

```
if (x == "0") ...
```

The type of a field is determined by context when possible; for example,

```
$1++
```

clearly implies that `$1` is to be numeric, and

```
$1 = $1 "," $2
```

implies that `$1` and `$2` are both to be strings. Coercion is done as needed.

In contexts where types cannot be reliably determined, for example,

```
if ($1 == $2) ...
```

the type of each field is determined on input. All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value `""` ; they are not numeric. Non-existent fields (that is, fields past `NF`) are treated this way, too.

As it is for fields, so it is for array elements created by `split`.

Mentioning a variable in an expression causes it to exist, with the value `""` as described above. Thus, if `arr[i]` does not currently exist,

```
if (arr[i] == "") ...
```

causes it to exist with the value `""` so the `if` is satisfied. The special construction

```
if (i in arr) ...
```

determines if `arr[i]` exists without the side effect of creating it if it does not.

14
Managing File Systems Securely

14
Managing File Systems Securely

Introduction

This chapter describes the access control mechanisms used to provide security for data contained in files. Because almost all of your activities on the system involve access to files or directories, you will deal with these access control mechanisms any time you use the computer system.

Because the UNIX operating system is a multi-user system, you usually do not work alone in the file system. All users of the system can follow path names to various directories and read and use files belonging to one another, as long as they have permission to do so. Without some form of access control all users could read all files, and it would be impossible to maintain the security of the data in the files. To provide data security, the operating system contains two access control mechanisms: Discretionary Access Control (DAC) and Mandatory Access Control (MAC). You must pass both sets of access checks to read or modify a file or to execute a program.

If you own a file, DAC allows you to 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. When you grant execute permissions for a directory, you allow the specified users to change directory to it. An administrator with the appropriate privilege can override the DAC restrictions set by the owner of a file.

MAC is controlled by the system (as configured by the system administrator), and restricts a user's access to data based on the sensitivity and topics associated with the data and the user. The owner of a file has no control over the MAC restrictions on the file.

This chapter describes how MAC works for files, directories, and devices. DAC is described in detail in Chapter 4, "*Using the File System*" of this guide.

Mandatory Access Control

Mandatory Access Control (MAC) is a security mechanism enforced by the Trusted Computing Base (TCB). (See the preface to this guide for a discussion of the TCB.) It is based on security levels (defined below) which are assigned to user processes and to objects such as files and directories. The system administrator assigns allowable login levels to users. The system compares your login level to the levels of the files and directories you try to access to decide which ones you can read or change. Thus it provides a way for the system to automatically restrict access to data.

While you set the Discretionary Access Controls (DAC) for your own files, MAC, as the name implies, is a mandatory control. The TCB, rather than the owner, sets the security level of a file or other object when the object is created. Only an administrator can change an object's level.

When you try to read, execute, or modify a file, the system enforces both MAC and DAC. It grants you access only if you pass both sets of controls.

The system administrator may assign you a single security level or several. Each time you log in, you work at only one of your levels. To change levels, you need to log out and log in again.

Since MAC is controlled by the TCB rather than the user, this section mainly explains how MAC limits what you can do on the system. If you have been assigned a single level, you may just want to skim this chapter to understand these limitations. If you are authorized to log in at more than one level, you will need to understand MAC in some detail to help you organize your files at different levels.

This section begins with definitions and examples of basic MAC concepts. This is followed by an overview of the MAC policies the TCB applies when comparing levels. The final section contains some suggestions for handling your files and directories if you are authorized to log in at multiple levels.

Subjects and Objects

The TCB uses MAC to control access to `objects` by `subjects.`

An object is something that contains or receives information. The system contains the following types of objects:

- files
- directories
- named pipes
- device special files
- symbolic links
- shared memory
- message queues
- semaphores
- processes (as targets of signals).

This section discusses MAC in terms of your access to files and directories. MAC access restrictions on other objects are similar, and differences will be noted in the following sections.

A subject is something that causes information to flow within the system. Loosely speaking, you can think of users and the programs they run as subjects. Technically, all subjects are `processes`. Each time you log in to the system or run a program, you create a process that is considered a subject by MAC.

Each subject and object has a label indicating its security level. MAC decides whether to grant or deny you access to a file, directory, or other object by comparing subject and object security levels. The next section defines security level.

Security Levels

A security `level` is a combination of a hierarchical `classification` and a set of zero or more non-hierarchical `categories`.

A classification indicates degree of sensitivity, with classifications going from lower degrees of sensitivity to higher ones. A few standard classifications are predefined on the system. The system administrator defines additional classifications. In one organization, material might have classifications such as Restricted (lowest), Secret (middle), and Top Secret (highest). At another site, there might be numerous distinct classifications, ranging from Open (lowest) through Confidential (middle) to Proprietary (highest). In each case, classifications are ordered from low to high.

A category is a grouping by topic. Several categories are predefined on the system. The system administrator defines additional categories reflecting the types of data stored on the system. For example, an organization could have categories for management, engineering, and sales. Another might define MiddleEast, EastBlock, and CommonMarket as categories. At some sites, the topics themselves might be confidential information, and categories would be given code names like Sundance or ProjectX.

Categories are unordered. They serve to keep information separated by topic. You can read and write data only in the categories you are authorized to access, as you will see in the *"MAC Security Policy"* section of this chapter. The administrator assigns each category a number for the system to use, but these numbers are arbitrary.

The administrator defines levels as combinations of classifications and categories; in addition, several levels are predefined. Each level has one classification, and may or may not contain one or more categories. The name of a level is written in the format

```
classification:category1,category2,...categoryn
```

For example, the following would be a valid level name:

```
Secret:Management,Sales
```

The administrator defines levels based on how the system is used. For example, if both proprietary and open sales data are kept on the system, the levels defined might include:

```
Proprietary:Sales
```

and

```
Open:Sales
```

There is a predefined level for files all users of the system need to read or execute:

```
SYS_PUBLIC
```

This level consists of one of the lowest defined classifications, `system`. It contains no categories, since categories are used to limit which users can access data.

Some projects require users to combine data from two or more categories. To allow this, the administrator will define one or more levels with appropriate sensitivity classifications and combinations of categories. For example, a user who logs in at the following level:

```
Proprietary:Engineering,Management,Sales
```

can read data at or below the `Proprietary` classification in each or any combination of the listed categories, but can write data only at the specified login level.

Only the classification/category combinations that have been predefined or specified by the system administrator are valid levels. Possible combinations that have not been defined are not valid. For example, the level

```
Open:Engineering,Management,Sales
```

would probably not be defined as a level if the system will not be used for data of low sensitivity that involves all three categories or if no one requires access to all three categories at the same time.

A level name that contains several categories can be quite lengthy, and therefore difficult to remember or to type. An administrator can assign a `level alias` as a shorthand level name to make it easier for you to specify a level.

To display a list of the security levels, classifications, and categories on your system, use the **`lvlprt`** command. The command displays three lists:

1. Under the heading `Levels:`, each level that is defined on the system is listed on a separate line. The level's alias, if it has one, appears at the beginning of the line, followed by two colons. Then the level's full name is displayed (the classification and optional list of categories, separated by a colon).
2. Under the heading `Classifications:`, each classification that is defined on the system is listed on a separate line. The classification number is displayed, followed by a colon and the classification name. Classifications are displayed in order, from lowest to highest.
3. Under the heading `Categories:`, the categories are displayed, one per line. Categories are displayed in the same format as classifications; that is, the category number is displayed followed by the category name. In the case of categories, the numbers do not indicate any hierarchical relationship.

Screen 14-1 shows an example of the output displayed by **`lvlprt`**.

```
$ lvlprt
Levels:
SYS_PUBLIC::system
SYS_PRIVATE::system:private
SYS_RANGE_MAX::range_max:ALL
USER_PUBLIC::user
USER_LOGIN::user:login
SYS_AUDIT::system:private,audit
SYS_OPERATOR::system:private,operator
SYS_RANGE_MIN::range_min
NotSecret
Green::NotSecret:proj_43,syseng
Red::NotSecret:proj_43
Blue::TopSecret:proj_43
Yellow::NotSecret:syseng

Classifications:
1:range_min
2:system
4:user
100:NotSecret
250:TopSecret
256:range_max

Categories:
1:private
2:audit
3:login
4:operator
100:syseng
143:proj_43
$
```

Screen 14-1. Example of Levels, Classifications, and Categories

Thirteen levels are shown in Screen 14-1. The first eight are the predefined levels that come with the system, while the last five levels have been defined by the administrator.

The first level defined by the administrator, `NotSecret`, consists of a classification with no categories. Since the full name of the `NotSecret` level is short, the administrator has chosen not to assign it an alias.

Each remaining level has been assigned an alias. The aliases in Screen 14-1 are `SYS_PUBLIC`, `SYS_PRIVATE`, `SYS_MAX_RANGE`, `USER_PUBLIC`, `USER_LOGIN`, `SYS_AUDITOR`, `SYS_OPERATOR`, `SYS_RANGE_MIN`, `Green`, `Red`, `Blue`, and `Yellow`. For each of these levels, the alias is displayed at the beginning of a line and separated from the full level name by two colons.

Except for the levels `SYS_PUBLIC`, `USER_PUBLIC`, `SYS_RANGE_MIN`, and `NotSecret`, which contain no categories, each level name is in the format

```
    Classification[:category1[,category2[,...categoryn]]]
```

For example,

```
    NotSecret:proj43,syseng
```

is the full name of the level with the alias `Green`.

Not all combinations of valid classifications and valid categories are defined as levels. In the system represented by Screen 14-1, `TopSecret:syseng` is not defined as a level and cannot be assigned to users or files.

The list of levels in Screen 14-1 is followed by the list of classifications ordered from least sensitive to most sensitive. For each classification, **`lvlprt`** displays the classification number, a colon, and the name. Six classifications are defined; **`range_min`** is the least sensitive and **`range_max`** is the most sensitive.

The numbers displayed with the classification indicate a hierarchical order, with lower numbers indicating less sensitivity and higher numbers more sensitivity. On the system shown in Screen 14-1, the administrator could use a number from 101 to 249 to add a classification intermediate in sensitivity between `NotSecret` and `TopSecret`.

The last part of the **`lvlprt`** display shown in Screen 14-1 is the list of categories. There are six categories: `private`, `audit`, `login`, `operator`, `syseng`, and `proj43`. For each category, its number is displayed, followed by a colon and the category name. Unlike classification numbers, category *numbers* are arbitrary. There is no hierarchical relationship among categories.

For more information on using **`lvlprt`**, see the **`lvlprt (1)`** page in the online *Command Reference.*

Comparing Security Levels

As was mentioned earlier, the system decides whether or not to grant you MAC access to a file, directory, or other object by comparing the level of the subject (that is, the process you are running) with that of the object. For write access, MAC requires that the subject's and object's levels be equal. For read or execute access, the subject's level must `dominate` that of the object. This section explains what it means for the subject's level to equal or dominate the object's. The following section, *"MAC Security Policy,"* explains how MAC controls what you can do on the system based on security levels.

The subject's level is equal to the object's only if both consist of the same classification and identical lists of categories. For example, if a user process and a file are both at the level `TopSecret:proj43`, then the level of the process is equal to that of the file. Similarly, an alias is equal to the level it is assigned to. In Figure 14-1, for example, `Blue` is equal to `TopSecret:proj43`. Both are names of the same level; the first is its alias, the second its full name. Each level is equal only to itself.

The concept of domination is more complex. Two things must be true for one level to dominate another. First, the classification of the dominating level must be equal or higher than the dominated level's. Second, the set of categories for the dominating level must include all the categories of the dominated level. In other words, the dominated level's categories are a subset (not necessarily a `proper subset` as defined in set theory) of the dominating level's categories.

For example, compare the levels `Red` and `Blue` in Screen 14-1. `Blue` is the alias for a level with the classification `TopSecret`, the most sensitive classification on this system. `Red` has a less sensitive classification level, `NotSecret`. So `Blue` has a higher classification than `Red`, and fulfills the first criterion for domination.

To determine if Blue dominates Red, you also need to compare the two levels' sets of categories. Red contains only one category, proj43. This category is also included in the set for Blue; it happens to be the only category in the set. So Blue also fulfills the second criterion, and dominates Red.

Similarly, Green (NotSecret:proj43,syseng) also dominates Red. In this case, the two levels have the same classification, NotSecret, satisfying the first part of the definition. Green's set of categories, proj43,syseng, includes Red's only category, proj43.

Now compare the levels Blue (TopSecret:proj43) and Green (NotSecret:proj43,syseng) in Screen 14-1. Blue has a higher classification (TopSecret) than Green, but Blue does not dominate Green. This is because Green contains a category, syseng, which is not part of Blue's set of categories. The levels Blue and Green are disjoint; that is, neither dominates the other.

Notice that by the definition, each level dominates itself. Each level has the same classification as itself, satisfying part one of the definition. It contains all of its own categories, satisfying part two of the definition.

What about the level SYS_RANGE_MIN which does not contain any categories? It is dominated by all of the levels shown in Screen 14-1. Since the classification range_min is the least sensitive of the defined classifications, every level defined on the system has an equal or higher classification. Since this level has no categories, any possible level would satisfy the second criterion for domination.

MAC Security Policy

The system compares your security level to that of each object you try to access. You will be granted or denied requested types of access depending on whether your level equals or dominates the object's level.

Your security level is set when you log in. The administrator assigns one or more allowable security levels to each user. In any login session, you will be identified with only one of the levels you are authorized to use.

If you have been assigned more than one level, one is designated as your default level. Alternatively, you can specify one of your assigned levels when you log in. The level you specify, or your default level if you do not specify a level, is called your login level. (See the *“Login Procedure”* subsection of Chapter 3, “*Basics for UNIX System Users*” for specific directions for logging in.)

Your login level determines the level of objects that you create. Files, directories, processes and other objects you create inherit your login level. You should be careful to create files only at the appropriate level. If you create a file at the wrong level, only an administrator can change the level of a file or directory.

Your login level restricts what information you can read and modify and what programs you can run. You can modify (or write) only objects at a level equal to your login level and can read or execute only those at levels dominated by your login level. There is one exception to this rule: to send a signal to a process (for example with the **kill** command), the level of the receiving process must dominate that of the sending process. In all cases, you must also have appropriate DAC permissions.

For example, on a system with the levels shown in Screen 14-1, if you are logged in at the level Blue (TopSecret:proj43) you can edit only those files at that level. Any new files you create will also be at the Blue level.

You can read only those files with levels dominated by Blue. This includes files at the levels Blue, Red (NotSecret:proj43), and NotSecret. You can run a program only if the program file is at a level dominated by Blue.

While you are logged in at the Blue level, MAC will not allow you to access any files with levels not dominated by Blue. For example, you can neither read nor edit a file at the level NotSecret:proj43,syseng, even if you have DAC permissions to do so.

Because almost anything you do on the system involves reading, writing, or executing objects, most of your actions are controlled by MAC. For example, to copy the file **/file1** into a file named **newfile** in the **/usr/mydir** directory, you can use the following command:

```
$ cp /file1 /usr/mydir/newfile <RETURN>
```

You need to pass a series of MAC checks for this command to succeed.

1. To execute the **cp** command, you need to have execute permission on the executable file for the command, **/usr/bin/cp.** To have MAC execute access, your level must dominate the levels of **/, /usr, /usr/bin,** and **/usr/bin/cp.**
2. First, **cp** searches the root directory for the file **file1**. Searching a directory requires the same permissions as executing a program file, so your level must dominate that of the root directory for MAC to allow the search.
3. Your level must dominate the level of **file1** so that **cp** can read and copy the file.
4. Now **cp** searches the root directory for the **usr** subdirectory, and the **usr** directory for the **mydir** subdirectory. You need MAC search permission on each directory in the path, so your level must dominate the levels of the root and **/usr** directories.

   Your MAC level must also be within the *level range* of the file system that will contain the file that's to be created. The file system level ranges are set by the system administrator.
5. If these checks succeed, **cp** is ready to create **/usr/mydir/newfile.** Adding a file to a directory changes the directory's contents, and therefore is a write operation on the directory. Your level must be equal to that of **/usr/mydir** for MAC to allow this.

Your new copy of **/file1, /usr/mydir/newfile,** will inherit your login level.

Figure 14-1 shows part of a file system with directories and files at several levels. (The section *“Organizing Files at Multiple Levels,”* later in this chapter, explains how to create a subdirectory at a different level from the parent directory.)

To clarify how MAC controls your access to files and directories, let's look at what you can and cannot do at various login levels. To simplify, the example assumes that you have DAC read, write, and execute permissions on all the files and directories in the diagram.

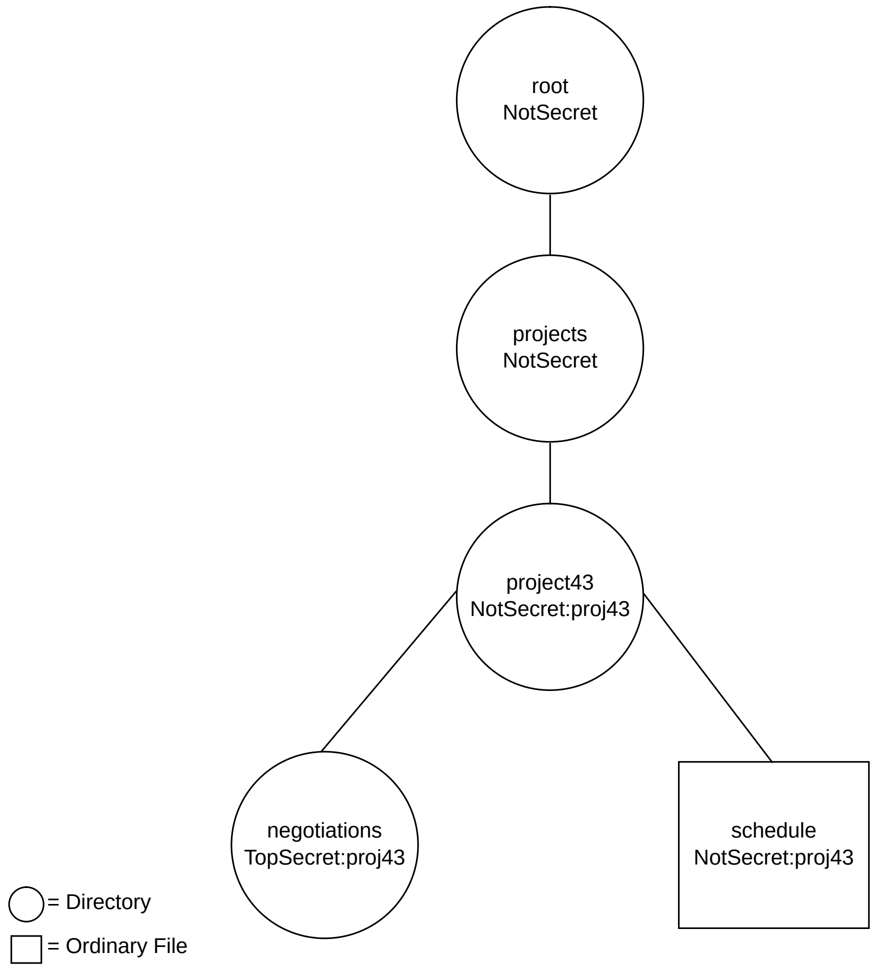


Figure 14-1. Directory Structure with Several Levels

Figure 14-1 shows the path to a directory (**/projects/project43** at level NotSecret:proj43) which contains a subdirectory (negotiations at level TopSecret:proj43) and a file (**schedule** at level NotSecret:proj43). The **root** and **projects** directories are at level NotSecret.

If you are logged in at NotSecret:proj43, you can modify only the files and directories at that level. You can edit the file **schedule** and can modify the directory project43 by creating a new file in that directory. You can't create a new file or subdirectory in projects, since your login level is not equal to its level.

You can read or execute a file or search or list a directory only if your level dominates its level. Since a level dominates itself, you can read the **schedule** file, which is at your login level. In Figure 14-1, your login level dominates the levels of all of the files and directories except the negotiations directory, which is at TopSecret:proj43.

Since your level does not dominate that of the negotiations directory, you can't display a list of the files in it or make it your current directory. You also can't read or modify any file in the negotiations directory, even if an administrator has changed its level to

NotSecret:proj43. This is because to read a file, you first have to search the directory the file is in.

If you log out and log in again at TopSecret:proj43, your new login level dominates the levels of all files and directories in the diagram. You now can search or list any directory and read any file shown in Figure 14-1. The only object shown that you can modify is the negotiations directory, since it is the only one at level TopSecret:proj43.

MAC prevents you from modifying objects at any level but your login level to keep you from accidentally making information available to unauthorized users. Any data that you enter when logged in at TopSecret:proj43 should be seen only by users whose login levels dominate that level. The MAC restriction protects both against users transferring information out of the appropriate security level and against "Trojan Horse" programs making sensitive data available to unauthorized users.

MAC restrictions also apply to files and other objects you create. Files you create inherit your login security level, as do processes you run. If you create a file, then log out and log in at a different security level, the MAC restrictions still apply to the file you created. You will not be able to modify it from a different security level. You can read it only if your new login level dominates the level at which you created it. If you give other users DAC permission to access your files, only those at appropriate security levels can do so.

Displaying Login and File Levels

If you forget at which level you are logged in, you can find out using the **-z** or **-Z** options of the **ps** command. This command displays information about processes which are dominated by your login level. Your login process is one of the processes displayed.

The **-Z** option displays the full level name for each displayed process. The **-z** option displays the level's alias if one has been assigned. If a level has no alias, the level identifier (LID), a number the system uses to identify the level, is displayed. For a detailed description of the **ps** command and its options, see the **ps(1)** page of the online *Command Reference.*

Screen 14-2 shows an example of the output of **ps** using the **-z** and **-Z** options.

```
$ ps -z
    PID TTY      TIME COMMAND LEVEL

   9980 tty12    0:00 ps      Blue
   9900 tty12    0:00 ksh     Blue
$ ps -Z
    PID TTY      TIME COMMAND LEVEL

   9980 tty12    0:00 ps      TopSecret:syseng
   9900 tty12    0:00 ksh     TopSecret:syseng
$
```

Screen 14-2. Example of Displaying Process Levels with ps

To display the level of a file or directory, use the **-z** option or the **-Z** option of the **ls** command. These are similar to the **-z** and **-Z** options of **ps**.

Screen 14-3 shows an example of displaying file levels. This is what the **ls** command would display if you were logged in at the TopSecret:proj43 level and working in the /projects/project43 directory of the file system shown in Figure 14-1.

```
$ ls -z
negotiations    Blue
schedule        Red
$ ls -Z
negotiations    TopSecret:proj43
schedule        NotSecret:proj43
$
```

Screen 14-3. Displaying File and Directory Levels with ls

You can display this information only for files and directories at levels dominated by your login level. If you log in at NotSecret:proj43, for example, **ls** will not display the level information for the negotiations directory.

Organizing Files at Multiple Levels

If you are authorized to log in at more than one security level, you should take special care to make sure that you are at the appropriate level for the data in the file when you create a file. Remember that the file will inherit your login level and that MAC protection depends on the file's security level.

NOTE

MAC protection of data depends on the level at which file containing the data is created. If you create a file while logged in at an inappropriate level for the data in that file, unintended users may be able to access the data, and intended users may be denied access.

If you accidentally create a file at the wrong security level, only an administrator can change the file's level. If the intended level dominates that at which you created the file, you can make a copy of the file at the correct level by logging in at that level, copying the file, then deleting the original file.

You should take into account the security levels available to you when organizing your subdirectories. Because you can only write in a directory at your login level and because files you create inherit your login level, each file you create will be at the same level as its parent directory. So you will need at least one directory for each login level at which you expect to create files.

Because of the tree structure of the file system, in order to have directories at the different login levels you are authorized to use, you need some way to create a subdirectory at a different level from its parent. You can do this with the **`-l`** option of the **`mkdir`** command, which lets you create a directory that is not at your login level.

To use this option, you must first log in at the level of the parent directory. You will modify the parent directory by adding the new subdirectory, and MAC requires that your login level be equal to the level of the object you modify.

For example, if you want to add a new subdirectory for sensitive data to the /projects/project43 directory shown in Figure 14-1, log in at NotSecret:proj43. Now use the **`cd`** command to make /projects/project43 your working directory.

MAC imposes two restrictions on the level of your new subdirectory:

- The new directory's level must dominate the level of the parent directory.
- The new directory's level must dominate your current login level.

When using **`mkdir -l`**, you should only create directories at levels at which you can log in. Otherwise, you will not be able to access the directory you create.

For example, if you are authorized to log in at TopSecret:proj43, you can create a subdirectory **`newdir`** at TopSecret:proj43 in /projects/project43.

Once you have created your new directory, the usual MAC restrictions apply. Since the new directory's level is not equal to or dominated by your login level, you need to logout and log in again at TopSecret:proj43 to create files in your new directory or display information about it.

Screen 14-4 shows the steps for creating a directory at a different level from its parent. Figure 14-2 shows the resulting directory structure.

Be careful in choosing a name for your new directory. If you are using the directory to keep files about a secret plan, you don't want the title to give away the plan. While only users at levels that dominate TopSecret:proj43 can access files within your new directory, anyone with access to the parent directory can see your directory's name. For example, if you create a directory **`MergerProposals`** at a level different from the parent directory's, users with access to the parent directory will know that a merger is being considered.

Managing Your Home Directory

If one of your login levels is dominated by all your other levels and if the administrator has given you a home directory at that level, you can use **`mkdir -l`** to set up your home directory. Following the instructions described in the previous section, you can create a subdirectory at each of your login levels in your home directory. If your login levels are disjoint (that is, if no one level is dominated by all others), you may need a system administrator's help in setting up your home directory.

```
login: -h NotSecret:proj43 user1
Password:
$ cd /projects/project43
$ pwd
/projects/project43
$ ls -Z
UX:ls:ERROR:cannot get level for file ./negotiations
schedule NotSecret:proj43
$ mkdir -l TopSecret:proj43 newdir
$ ls -Z
UX:ls:ERROR:cannot get level for file ./negotiations
UX:ls:ERROR:cannot get level for file ./newdir
schedule NotSecret:proj43
$ cd newdir
UX:cd:ERROR:newdir:Permission Denied
$ logout
login: -h TopSecret:proj43 user1
Password:
$ cd /projects/project43
$ ls -Z
negotiationsTopSecret:proj43
schedule NotSecret:proj43
newdir   TopSecret:proj43
$
```

Screen 14-4. Creating a Directory at a Different Level

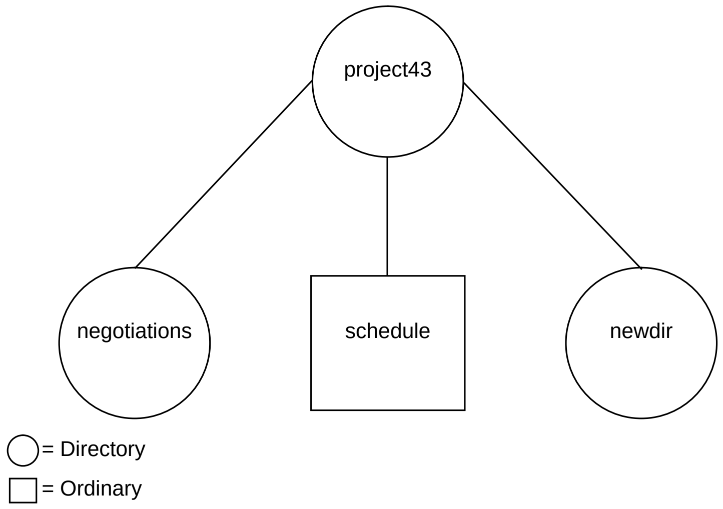


Figure 14-2. Directory Structure after Using mkdir `-l`

Multilevel Directories

The Mandatory Access Control (MAC) restrictions on the creation of files require that the user be at the same level as the directory in which the file is created. But there are some directories in which all users need to create files. For example, **`/tmp`** is used by editors and compilers to store temporary files. The editors and compilers run by users at different levels would need to create temporary files at different levels in violation of MAC restrictions on directories.

To solve this problem, the multilevel directory (MLD) is provided, in which any user can create files but files created by users logged in at other levels are "invisible." In fact, it's possible for users at different levels to create and use different files with the same name.

For example, a user at level A can create a temporary file **`/tmp/file1`**, and a user at level B can create a file with the same name. It is possible that the users will never know that there are two files named **`/tmp/file1`**.

The system transparently maps access to the multilevel directory to a subdirectory, called an effective directory, based on the user's level. The two users in the example above are actually creating files in different directories. The effective directories are individually created by the system when each one is initially accessed: the creation of an effective directory is automatic and transparent to the user.

The structure of multilevel directories is shown in Figure 14-3.

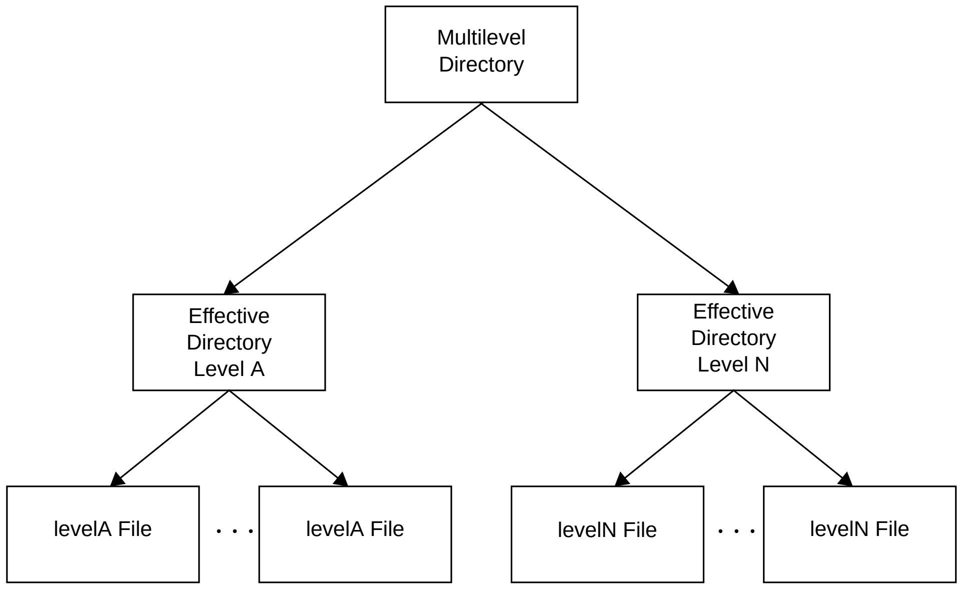


Figure 14-3. Structure of a Multilevel Directory

Associated with each process is a multilevel directory **`mode`** that determines the type of access to multilevel directories. This mode, which can be either `real` or `virtual`, is inherited from the parent process. The kernel uses the multilevel directory mode to determine how an access attempt to a multilevel directory should be handled.

Normally, user processes are in virtual mode. In virtual mode, if you try to access a multilevel directory, the system will bypass the directory and access the effective directory for your level.

If you log in at different MAC security levels at different times, files in a multilevel directory may seem to appear and disappear, or change contents as you change levels, because you will be accessing directories at different levels in the multilevel directory. Screen 14-5 shows an example of this.

```
login: -h LEVEL1 user1
Password:
$ echo foo > /multilevel/bar
$ ls /multilevel/bar
/multilevel/bar
$ logout
login: -h LEVEL2 user1
Password:
$ ls /multilevel/bar
/multilevel/bar: No such file or directory
$ echo bar > /multilevel/bar
$ ls /multilevel/bar
/multilevel/bar
$ cat /multilevel/bar
bar
$ logout
login: -h LEVEL1 user1
Password:
$ ls /multilevel/bar
/multilevel/bar
$ cat /multilevel/bar
foo
$
```

Screen 14-5. Accessing a Multilevel Directory from Different Levels

If an effective directory does not exist at your level, then the kernel creates it automatically. This can cause error messages that seem odd if you try to access a multilevel directory that is in a full file system or in one that is mounted read-only. Examples are shown in Screen 14-6.

```
$ cd /read-only
/read-only: cannot create
$ ls /read-only
/read-only: cannot create
$
$ cd /tmp
/tmp: No space left on device
$ ls /tmp
/tmp: No space left on device
$
```

Screen 14-6. Accessing a Multilevel Directory in a Read-only or Full File System

In real mode, the system treats a multilevel directory like a normal directory and the effective directories as normal subdirectories. If you need to see files in an MLD at a level other than your login level, you will need to change to real mode. The process in real mode can see all effective directories in the multilevel directory. The process can access all files in the effective directories, if the process passes the usual MAC and DAC checks (see the *"Mandatory Access Control"* and *"Discretionary Access Control"* sections in this chapter).

The **`mldmode`** command and the **`mldmode`** shell builtin let you switch between real and virtual mode, perform a single command in a specified mode, or find out the multilevel directory mode of your current process. For a detailed description of the command, see the **`mldmode(1)`** page of the online *Command Reference*. For a description of the shell builtin, see the **`sh(1)`** page of the online *Command Reference.*

Accessing Devices

The access restrictions that apply to files and directories (MAC and DAC) also apply to devices. However, an additional access control is placed on devices. This restriction requires that a device be `allocated` for public use before a non-privileged process can use it. Only privileged processes can allocate a device.

Some devices are usually allocated for general use when the system is initialized: `null`, `zero`, and `tty`. These devices can be accessed by all unprivileged processes that pass the MAC and DAC restrictions.

Other devices, such as tape drives, can be allocated for you by an administrator when appropriate. Check with your system administrator if you need to use a device that is not allocated for general use.

The `tty` device associated with your login session is allocated for you by the system during the login process.

Detailed information on device allocation can be found in the "Security" chapter of the *System Administrator's Guide*, and the "*Security Considerations*" chapter of the *Programming With UNIX System Calls and Application Release Notes.*

A
Summary of the File System

UNIX System Files

This appendix summarizes the description of the file system given in “What Is the UNIX System” chapter, and reviews the major system directories in the root directory.

File System Structure

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.

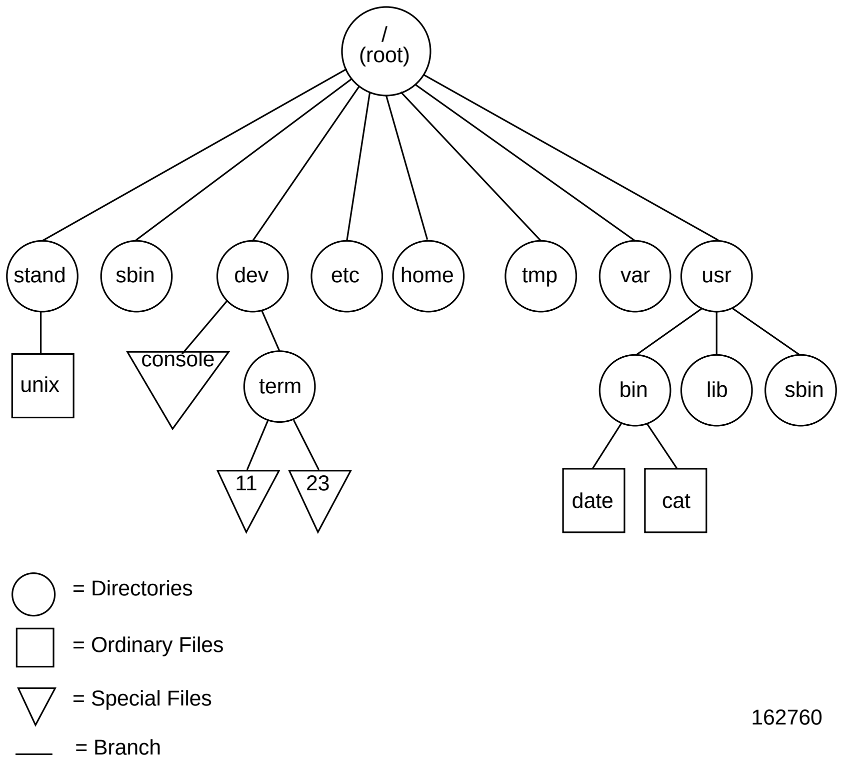


Figure A-1. Directory Tree from root

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 V Release 4 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 listing of the directories and files in the root directory on your UNIX system, enter 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 **/usr/bin** (which contains UNIX system executable files) by entering the following command line:

```
$ cd /usr/bin RETURN
```

To list the contents of a directory, enter

```
$ ls RETURN
```

for the short format listing, or

```
$ ls -l RETURN
```

for the long format listing

To list the contents of a directory in which you are not located, say **/usr/bin**, enter

```
$ ls /usr/bin RETURN
```

for the short format listing, or

```
$ ls -l /usr/bin RETURN
```

for the long format 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 the root directory).
/stand	contains programs and data files used in the booting process.
/sbin	contains essential executables used in the booting process and in manual system recovery.

`/dev`	contains special files that represent peripheral devices, such as:

```
console console
lp      line printer
term/*  user terminal(s)
dsk/*   disks
```

`/etc`	contains machine-specific administrative configuration files and system administration databases.
`/home`	the root of a subtree for user directories.
`/tmp`	contains temporary files, such as the buffers created for editing a file.
`/var`	the root of a subtree for varying files such as log files.
`/usr`	contains other directories, including `lib` and `bin`.
`/usr/bin`	contains many executable programs and utilities, including the following:

```
cat
date
login
grep
mkdir
who
```

`/usr/lib`	contains libraries for programs and languages.

B

Summary of UNIX System Commands

Basic UNIX System Commands

at	Request that a command be run in background mode at a time you specify on the command line. A sample format is: at 8:45am Jun 09 <RETURN> *command1* <RETURN> *command2* <RETURN> <CTRL><d> If you use the **at** command without the date, the command executes within twenty-four hours of the time specified.
banner	Display a message (in words up to ten 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 timesaving if you have a set of commands you frequently submit using this command.
cat	Display the contents of a specified file at your terminal. To halt the output on an ASCII terminal temporarily, use <CTRL><s>; type <CTRL><q> to restart the output. To interrupt the output and return to the shell on an ASCII terminal, press the <BREAK> or <DELETE> key.
cd	Change directory from the current one to your home directory. If you include a directory name, this command changes from the current directory to the directory specified. By using a path name in place of the directory name, you can jump several levels with one command.
chmod	Set the mode of a file(s) or directory. You must be the file owner. You can set additional access permissions with the **setacl** command.

cp	Copy a specified file into a new file, leaving the original file intact.
cut	Cut out specified fields from each line of a file. This command can be used to cut columns from a table, for example.
date	Display the current date and time.
diff	Compare two files. The **diff(1)** command reports which lines are different and what changes should be made to the second file to make it the same as the first file.
echo	Display input on the standard output (the terminal), including the <RETURN>, and returns a prompt.
ed	Edit a specified file using the line editor. If there is no file by the name specified, the **ed(1)** command creates one. See Chapter 6, the “*Line Editor (ed) Tutorial*” for detailed instructions on using the **ed(1)** editor.
find	Search for files below a specified path which meet specified criteria. See the **find(1)** page of the *Command Reference* for details.
getacl	Display the owner, group, and Access Control List of files or directories. For a directory, the default ACL for files created in the directory is also displayed.
grep	Search a specified file(s) for a specified 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 *Compilation Systems Manual* for details.
lp	Print the contents of a specified file on a line printer, giving you a paper copy of the file.
lpstat	Display the status of any requests made to the line printer. Options are available for requesting more detailed information.
ls	List the names of all files and directories except those whose names begin with a dot (**.**). Options are available for listing more detailed information about the files in the directory. (See the **ls(1)** page in the *Command Reference* for details.)
lvlprt	Display a list of the security levels, categories, and classifications defined on the system.
mail	Display any electronic mail you may have received at your terminal, one message at a time. Only messages at your current security level will be displayed. 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. The message will inherit your current security level. 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 the <`BREAK`> key to interrupt the mail session.
`mailcheck`	Check if you have mail at levels dominated by your current level.
`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 online *Operating System API Reference* 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, it is convenient to move into the new directory with the **`cd`** command.
`mldmode`	Display or change the `multilevel directory mode` of your current process. When you access a multilevel directory in `virtual` mode, you see only the contents of the `effective directory` at your current security level. In `real` mode, you can see any effective directories at levels you dominate. Normally, you should operate in `virtual` mode.
`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 of the system. Error and output 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`	The **`ps(1)`** command does not show the status of jobs in the **`at(1)`** or **`batch(1)`** queues, but it includes these jobs when they are executing.
`pwd`	Display the full path name of the current working directory.

`rm`	Remove a file from the file system. You can use metacharacters with the **`rm(1)`** command but should use them with caution; a removed file cannot be recovered easily.
`rmdir`	Remove a directory. You cannot be in the directory you want to delete. Also, the command 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 the **`-r`** option of **`rm(1)`** in the *Command Reference* for removing directories that are not empty.)
`setacl`	Create or change the Access Control List of a file or directory, or the default ACL of a directory. You can use the **`setacl`** command only for files and directories you own.
`sort`	Sort a file in ASCII order and display the results on your terminal. ASCII order is as follows: 1. special characters 2. numbers before letters 3. upper case before lower case 4. 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 *Command Reference.*
`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.
`stty`	Report the settings of certain input/output options for your terminal. When issued with the appropriate options and arguments, **`stty(1)`** also sets these input/output options. (See the **`stty(1)`** page in the *Command Reference.)*
`tcpio`	Backup files to an archive on removable media and restore files from the archive. (See the **`tcpio(1)`** page in the *Command Reference* for details.) For security reasons, some sites restrict access to archive devices to administrators.
`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 online *Command Reference* 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 prompts you (with a `?`) to take action.

`uustat`	Report the status of the **`uuto(1)`** command you issued to send files to another user.
`uuto`	Send a specified file to another user. Specify the destination in the format *system*!*login*. The *system* must be on the list of systems generated by the **`uuname(1C)`** 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 7, the "*Screen Editor (vi) Tutorial*" chapter 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. Character counts are provided in bytes.
`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 *Compilation Systems Manual* for details.

C
Quick Reference to ed Commands

ed Quick Reference

The general format for **ed** commands is:

`[`*address1*`,`*address2*`]`*command*`[`*parameter*`]. . . <RETURN>`

where *address1* and *address2* denote line addresses and the *parameters* show the data on which the command operates. The commands appear on your terminal as you type them. You can find complete information on using **ed** commands in the “Line Editor (ed) Tutorial” chapter.

The following is a glossary of **ed** commands. The commands are grouped according to function. Refer to the following table:

Overview of Commands for Getting Started with ed

Table C-1. ed Command Overview

ed *filename*	Accesses the **ed** line editor to edit a specified file.
a	Appends text after the current line.
.	Ends the text input mode and returns to the command mode.
p	Displays the current line.
d	Deletes the current line.
<RETURN>	Moves down one line in the buffer.
-	Moves up one line in the buffer.
w	Writes the buffer contents to the file currently associated with the buffer.
q	Ends an editing session. If changes to the buffer are not written to a file, a warning (`?`) is issued. Typing `q` a second time ends the session without writing to a file.

Overview of ed Line Addressing Commands

`1, 2, 3...`	Denotes line addresses in the buffer.
`.`	Denotes address of the current line in the buffer.
`.=`	Displays the current line address.
`$`	Denotes the last line in the buffer.
`,`	Addresses the first through the last line.
`;`	Addresses the current line through the last line.
`+x`	Relative address, determined by adding `x` to the current line number.
`-x`	Relative address, determined by subtracting `x` from the current line number.
/abc	Searches forward in the buffer and addresses the first line after the current line that contains the pattern *abc*.
?abc	Searches backward in the buffer and addresses the first line before the current line that contains the pattern *abc*.
`g`*/abc*	Addresses all lines in the buffer that contain the pattern *abc*.
`v`*/abc*	Addresses all lines in the buffer that do not contain the pattern *abc*.

Overview of ed Display Commands

`p`	Displays the specified lines in the buffer.
`n`	Displays the specified lines preceded by their line addresses and a tab space.

Overview of ed Text Input Commands

`a`	Enters text after the specified line in the buffer.
`i`	Enters text before the specified line in the buffer.
`c`	Replaces text in the specified lines with new text.
`.`	When typed on a line by itself, ends the text input mode and returns to the command mode.

Overview of ed Delete Text Commands

d	Deletes one or more lines of text (command mode).
u	Undoes the last command given (command mode).

Overview of ed Substitute Text Commands

address1,address2s/old_text/new_text/command

Substitutes *new_text* for *old_text* within the range of lines denoted by *address1,address2* (which may be numbers, symbols, or text). The *command* may be **g**, **l**, **n**, **p**, or **gp**.

Overview of ed Special Pattern-matching Characters

`.`	Matches any single character.
`*`	Matches zero or more occurrences of the preceding character.
`[...]`	Matches any character that is in the brackets.
`[^...]`	Matches any character that is not in the brackets.
`.*`	Matches zero or more occurrences of any character.
`^`	Matches the beginning of the line.
`$`	Matches the end of the line.
`\`	Takes away the meaning of the special character that follows.
`&`	Substitutes the text matched by the substitution pattern in the replacement string.
`%`	Repeats the last replacement string.

Overview of ed Text Movement Commands

m	Moves the specified lines of text after a destination line; deletes the lines at the old location.
t	Copies the specified lines of text and places the copied lines after a destination line.
j	Joins contiguous lines.
w	Copies (writes) the buffer contents into a file.
r	Reads in text from another file and appends it to the buffer.
W	Appends text to an existing file.

Overview of Other Useful ed Commands and Information

`h`	Displays a short explanation for the preceding diagnostic response (`?`).
`H`	Turns on the help mode, which automatically displays an explanation for each diagnostic response (`?`) during the editing session.
`l`	Displays nonprinting characters in the text.
`f`	Displays the current file name.
`f` *newfile*	Changes the current file name associated with the buffer to *newfile*.
`!`*command*	Allows you to escape, temporarily, to the shell to execute a shell command.
`ed.hup`	Saves the buffer in a special file called **`ed.hup`** if **`ed`** is interrupted.

D
Quick Reference to vi Commands

vi Quick Reference

This appendix is a glossary of commands for the screen editor, **`vi`**. The commands are grouped according to function.

The general format of a **`vi`** command is:

> *[x][command]text-object*

where *x* denotes a number and *text-object* shows the portion of text on which the command operates. For an introduction to the use of **`vi`** commands, see Chapter 7, the "*Screen Editor (vi) Tutorial*" chapter.

Overview of Shell Commands Used with vi

`TERM=`*code*	Puts a code name for your terminal into the variable `TERM`.
`export` `TERM`	Conveys the value of `TERM` (the terminal code) to any UNIX system program that is terminal dependent.
`tput init`	Initializes the terminal so that it will function properly with various UNIX system programs.

NOTE

Before you can use **`vi`**, you must complete the first three steps represented by the above lines: setting the `TERM` variable, exporting the value of `TERM`, and running the **`tput init`** command.

`vi` *filename*	Accesses the **`vi`** screen editor so that you can edit a specified file.

Overview of Basic vi Commands

Table D-1 lists and describes the basic vi commands:

Table D-1. Basic vi Commands

a	Enters text input mode and appends text after the cursor.
<ESC>	Escape; leaves text input mode and returns to command mode.
h	Moves the cursor one character to the left.
j	Moves the cursor down one line in the same column.
k	Moves the cursor up one line in the same column.
l	Moves the cursor one character to the right.
x	Deletes the current character.
<RETURN>	Moves the cursor down to the beginning of the next line.
ZZ	Writes to the file those changes made to the buffer that have not already been written and quits **`vi`**.
:w	Writes to the file those changes made to the buffer.
:q	Quits **`vi`** if changes made to the buffer have been written to a file.

Overview of vi Commands for Positioning by Character

Table D-2 lists and describes the vi cursor positioning commands.

Table D-2. vi Cursor Positioning Commands

h	Moves the cursor one character to the left.
<BACKSPACE>	Backspace; moves the cursor one character to the left.
l	Moves the cursor one character to the right.
<SPACEBAR>	Moves the cursor one character to the right.
f*x*	Moves the cursor right to the specified character *x*.
F*x*	Moves the cursor left to the specified character *x*.
t*x*	Moves the cursor right to the character just before the specified character *x*.

Table D-2. vi Cursor Positioning Commands (Cont.)

T*x*	Moves the cursor left to the character just after the specified character *x*.
;	Continues the search for the character specified by the **`f`**, **`F`**, **`t`**, or **`T`** commands. The **`;`** remembers the character specified and searches for the next occurrence of it on the current line.
,	Continues the search for the character specified by the **`f`**, **`F`**, **`t`**, or **`T`** commands. The **`,`** remembers the character specified and searches for the previous occurrence of it on the current line.

Overview of vi Commands for Positioning by Line

Table D-3 lists and describes the vi line positioning commands.

Table D-3. vi Line Positioning Commands

j	Moves the cursor down in the same column one line from its present position.
k	Moves the cursor up in the same column one line from its present position.
+	Moves the cursor down to the beginning of the next line.
<RETURN>	Moves the cursor down to the beginning of the next line
-	Moves the cursor up to the beginning of the next line.

Overview of vi Commands for Positioning by Word

`w`	Moves the cursor to the right to the first character in the next word.
`b`	Moves the cursor back to the first character of the previous word.
`e`	Moves the cursor to the end of the current word.

Overview of vi Commands for Positioning by Sentence

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

Overview of vi Commands for Positioning by Paragraph

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

Overview of vi Commands for Positioning in the Window

`H`	Moves the cursor to the first line on the screen, or "home."
`M`	Moves the cursor to the middle line on the screen.
`L`	Moves the cursor to the last line on the screen.

Overview of vi Commands for Scrolling

Table D-4 lists and describes the vi scrolling commands.

Table D-4. vi Scrolling Commands

<CTRL><f >	Scrolls the screen forward a full window, revealing the window of text below the current window.
<CTRL><d>	Scrolls the screen down a half window, revealing lines of text below the current window.
<CTRL><b>	Scrolls the screen back a full window, revealing the window of text above the current window.
<CTRL><u>	Scrolls the screen up a half window, revealing the lines of text above the current window.

Overview of vi Commands for Positioning on a Numbered Line

`G`	Moves the cursor to the beginning of the last line in the buffer.
*n***`G`**	Moves the cursor to the beginning of the *n*th line of the file (*n* = line number).

Overview of vi Commands for Searching for a Pattern

`/`*pattern*	Searches forward in the buffer for the next occurrence of the pattern of text. Positions the cursor under the first character of the pattern.

?*pattern*	Searches backward in the buffer for the first occurrence of pattern of text. Positions the cursor under the first character of the pattern.
n	Repeats the last search command.
N	Repeats the search command in the opposite direction.

Overview of vi Commands for Inserting Text

Table D-5 lists and describes the vi text insertion commands.

Table D-5. vi Text Insertion Commands

a	Enters text input mode and appends text after the cursor.
i	Enters text input mode and inserts text before the cursor.
o	Enters text input mode by opening a new line immediately below the current line.
O	Enters text input mode by opening a new line immediately above the current line.
<ESC>	Escape; returns to command mode from text input mode (entered with any of the above commands).

Overview of vi Commands for Deleting Text

Table D-6 lists and describes the vi deleting commands.

Table D-6. vi Deleting Commands

Text Input Mode:	
<BACKSPACE> <ESC>	Backspace; deletes the current character.
<CTRL> <w> <ESC>	Deletes the current word delimited by blanks.
@ (Kill Character) <ESC>	Erases the current line of text.

Command Mode:

x	Deletes the current character.
dw	Deletes a word (or part of a word) from the cursor through the next space or to the next punctuation.
dd	Deletes the current line.
*n***d***x*	Deletes *n* number of text objects of type *x*, where *x* may be as a word, line, sentence, or paragraph.

`D`	Deletes the current line from the cursor to the end of the line.

Overview of vi Commands for Modifying Text

`r`	Replaces the current character.
`s`	Deletes the current character and appends text until the `<ESC>` command is typed.
`S`	Replaces all the characters in the current line.
`~`	Changes uppercase to lowercase or lowercase to uppercase.
`cw`	Replaces the current word or the remaining characters in the current word with new text, from the cursor to the next space or punctuation.
`cc`	Replaces all the characters in the current line.
*n*c*x*	Replaces *n* number of text objects of type *x*, where *x* may be a word, line, sentence, or paragraph.
`C`	Replaces the remaining characters in the current line, from the cursor to the end of the line.

Overview of Cutting and Pasting Text with vi

`p`	Places the contents of the temporary buffer (containing the output of the last delete or yank command) into the text after the cursor or below the current line.
`yy`	Yanks (extracts) a specified line of text and puts it into a temporary buffer.
*n*y*x*	Extracts a copy of *n* number of text objects of type *x* and puts it into a temporary buffer.
	`Places a copy of text object` *x* into a register named by a letter *l*. *x* may be a word, line, sentence, or paragraph.
	`Places the contents of register` *x* after the cursor or below the current line.

Overview of Special vi Commands

Table D-7 lists and describes special vi commands.

Table D-7. Special vi Commands

<CTRL><q>	Gives the line number of current cursor position in the buffer and modification status of the file.
.	Repeats the action performed by the last command.
u	Undoes the effects of the last command.
U	Restores the current line to its state prior to present changes.
J	Joins the line immediately below the current line with the current line.
<CTRL><l>	Clears and redraws the current window.

Overview of ex Line Editor Commands Used with vi

Table D-8 lists and describes ex commands used with vi.

Table D-8. ex Commands Used with vi

:	Tells **vi** that the next commands you issue will be line editor commands.
:sh	Temporarily returns to the shell to perform some shell commands without leaving **vi**.
<CTRL><d>	Escapes the temporary return to the shell and returns to **vi** so you can edit the current window.
:*n*	Goes to the *n*th line of the buffer.
:x,zw filename	Writes lines from the number *x* through the number *z* into a new file called *filename*.
:$	Moves the cursor to the beginning of the last line in the buffer.
:.,$d	Deletes all the lines from the current line to the last line.
:r *filename*	Inserts the contents of the file *filename* under the current line of the buffer.
:s/*text*/*new_text*/	Replaces the first instance of *text* on the current line with *new_text*.
:s/*text*/*new_text*/g	Replace every occurrence of *text* on the current line with *new_text*.
:g/*text*/s//*new_text*/g	Changes every occurrence of *text* in the buffer to *new_text*.

Overview of Commands for Quitting vi

Table D-9 lists and describes ex commands for quitting vi.

Table D-9. Commands for Quitting vi

`ZZ`	Writes the buffer to the file if you haven't already done so, and quits **`vi`**.
`:wq`	Writes the buffer to the file and quits **`vi`**
`:w` *filename* **`:q`**	Writes the buffer to the new file *filename* and quits **`vi`**.
`:w!` *filename* **`:q`**	Overwrites the existing file *filename* with the contents of the buffer and quits **`vi`**.
`:q!`	Quits **`vi`** whether or not changes made to the buffer were written to a file. Does not incorporate changes made to the buffer since the last write (**`:w`**) command.
`:q`	Quits **`vi`** if changes made to the buffer were written to a file.

Overview of Special Options for vi

Table D-10 lists and describes special options for vi.

Table D-10. Special Options for vi

`vi` *file1 file2 file3*	Enters three files into the **`vi`** buffer to be edited. Those files are *file1*, *file2*, and *file3*.
`:w` **`:n`**	When more than one file has been called on a single **`vi`** command line, writes the buffer to the file you are editing and then calls the next file in the buffer (use **`:n`** only after **`:w`**).
`vi -r` *file1*	Restores the changes made to *file1* that were lost because of an interrupt in the system.
`view` *file1*	Displays *file1* in the read-only mode of **`vi`**. Any changes made to the buffer will not be allowed to be written to the file.

E
Summary of Shell Command Language

Summary of Shell Command Language

This appendix is a summary of the shell command language discussed in Chapter 9, "*Programming with the UNIX System Shell*". The first section reviews metacharacters, special characters, input and output redirection, variables, and processes. These are arranged by topic in the order that they were discussed in the chapter. The second section contains models of the shell programming constructs.

The Vocabulary of Shell Command Language

Special Characters in the Shell

`*\ ?\ [ ]`	Metacharacters; used to provide a shortcut to referencing filenames, through pattern matching.
`&`	Executes commands in the background mode.
`;`	Sequentially executes several commands typed on one line, each pair separated by `;`.
`\`	Turns off the meaning of the immediately following special character.
`´...´`	Enclosing single quotes turn off the special meaning of all characters except single quotes.
`"..."`	Enclosing double quotes turn off the special meaning of all characters except `$`, single quotes, and double quotes.

Redirecting Input and Output

`<`	Redirects the contents of a file into a command.
`>`	Redirects the output of a command into a new file, or replaces the contents of an existing file with the output.
`>>`	Redirects the output of a command so that it is appended to the end of a file.

|	Directs the output of one command so that it becomes the input of the next command.
`command`	Substitutes the output of the enclosed command in place of *`command`*.

Executing and Terminating Processes

`batch`	Submits the following commands to be processed at a time when the system load is at an acceptable level. `<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 out.

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 automatically assigned by the shell from the arguments of the command line invoking the shell 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.
`MAIL`	Gives the name of the file containing your electronic mail.
`PS1, PS2`	Defines the primary and secondary prompt strings, respectively.
`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.

F
Setting up the Terminal

Setting the TERM Variable

Because some commands are terminal dependent, the system must know what type of terminal you are using whenever you log in. The system determines the characteristics of your terminal by checking the value of a variable called `TERM` which holds the name of a terminal. If you 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, enter the command lines shown, 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; otherwise, they will not work. Also, this procedure must be repeated every time you log in. Therefore, most users put these lines into a file called **`.profile`** that is automatically executed every time they log in.

The first two lines of input tell the UNIX system shell what type of terminal you are using. The **`tput init`** command line instructs your terminal to behave in ways that 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(1)`** command uses the entry corresponding to your terminal in its 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 the **`/usr/share/lib/terminfo`** directory, depending on the system.

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 online *Operating System API Reference.*

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 type is recognized by the system.

You must also verify that the name you put into the `TERM` variable is a recognized terminal name. There are usually at least two recognized names: the name of the manufacturer and the model number. However, there are several ways to represent these names: by varying the use of uppercase and lowercase, using abbreviations, and so on. 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. Enter

```
$tput -Tterminal_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 the **`/usr/share/lib/terminfo`** directory. This directory contains 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. (This name can be either a letter, such as the initial A in AT&T, or a number, such as the initial 5 in 5425.)

Find the file whose name matches the first character of the name of your terminal. Then list the contents of the file 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.

For example, suppose your terminal is a Model wyse150. Your login is `jim` and you are currently in your home directory.

1. First, verify that your system supports your terminal by running the **`tput`** command.
2. Next, find an acceptable name for it in the **`/usr/share/lib/terminfo/a`** directory.

The following screen shows which commands you need to do this:

```
$ tput -wyse150 longname <RETURN>
wyse150
$ cd /usr/share/lib/terminfo/a <RETURN>
$ ls *150 <RETURN>
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
wyse150
```

Now you are ready to put the name you found, `wyse150`, in the `TERM` variable. Whenever you do this, you must also export `TERM` and execute **`tput init`**.

```
$ TERM=wyse150 <RETURN>
$ export TERM <RETURN>
$ tput init <RETURN>
```

The UNIX system now knows what type of terminal you are using and will execute commands appropriately.

G
Glossary

Glossary

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.”
associative array	A collection of data (an array) where individual items may be indexed (accessed) by a string, rather than by an integer as is common in most programming languages. The data item is said to be “associated” with the pair *array-name: string*, where *string* is the index.
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. Compare with “foreground.”
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.”
coerce	To force a data object to be treated a particular way.
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 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.”
concatenate	To combine two strings into one that comprises the characters of the first followed by the characters of the second.
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 control-s <CTRL><s>) stops the display so you can read it, and typing control-q <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.

Discretionary Access Control (DAC)	If you own a file, DAC allows you to 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. When you grant execute permissions for a directory, you allow the specified users to change directory to it. An administrator with the appropriate privilege can override the DAC restrictions set by the owner of a file. For more detail, see Chapter 4,“*Using the File System*”. Compare with “Mandatory Access Control (MAC).”
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*.”)
disk	A magnetic data storage device consisting of several round plates similar to phonograph records. Disks store large amounts of data and allow quick access to any piece of data.
electronic mail	The feature of an operating system that allows computer users to exchange written messages via the computer. The UNIX system **`mail`** and **`mailx`** commands 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. Compare with "background."
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," compare with "relative 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 control-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.
job identifiers	A value assigned to components of a remote administrative activity or to the entire remote task; there are three types of job identifiers: service job identifiers representing an application view of a remote operation, administrative job identifiers representing the invocation of an administrative command per destination machine, and primitive job identifiers representing the invocation of a remote administration primitive to or at a single destination machine.
line editor	An editing program in which text is operated upon on a line-by-line basis in 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.
machine alias	An abbreviated notation for a collection of remote machines; machine aliases can be used on command lines to facilitate the specifications of many destination machines.
Mandatory Access Control (MAC)	MAC is controlled by the system (as configured by the system administrator), and restricts a user's access to data based on the sensitivity and topics associated with the data and the user. The owner of a file has no control over the MAC restrictions on the file. For more detail, see Chapter 14, “*Managing Files Securely*”. Compare with “Discretionary Access Control (DAC).”
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.
multilevel directory (MLD)	A directory in which any user can create files under the "Mandatory Access Control (MAC)" mechanism, but files created by users logged in at other levels are "invisible." For more detail, see Chapter 14, "*Managing Files Securely*".
multiuser	The ability of an operating system to support several users on the system at the same time.
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 type of variable used in 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 `$0`. See “variable” and “shell procedure.”
primitives	A primitive operation that is the result of the decomposition of a remote administrative task for execution over a network service; examples are file transfer (ft), directory transfer (dt), and remote execution (re).
printer	An output device that prints the data it receives from the computer on paper.
process	Generally a program that is at some stage of execution. In the UNIX system, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current directory, status of files, information recorded at login time, and various other items.
program	The instructions given to a computer on how to do a specific task. Programs are user-executable software.
prompt	A cue displayed at your terminal by the shell, telling you that the shell is ready to accept your next request. The prompt can be a character or a series of characters. The UNIX system default prompt is the dollar sign character (`$`).

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. See “pathname,” compare with “full pathname.”
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 (`>`).
service identifier	A string supplied by an administrative command when it requests a remote operation primitive that associates the primitive with administrative services; a service identifier determines default file locations in the remote administration directory structure.
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. The **`sh(1)`** and **`ksh(1)`** entries in online *Command Reference* describe two of the available shells.
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 a program. Shell special characters are used for common shell functions such as file redirection, piping, background execution, and file name expansion. Shell special characters include `<`, `>`, `|`, `;`, `&`, `*`, `?`, `[`, and `]`. Editors such as **`ed`** and **`vi`** also have special characters.
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 error	An open file that is normally connected directly to a primary output device, such as a terminal printer or screen. Error messages and other diagnostic output normally goes to this file and then to the output device. You can redirect the standard error output into another file instead of to the printer or screen; use an argument in the form `2>` *file*. Error output will then go to the specified file.
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 on the 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.
Trusted Computing Base (TCB)	The totality of the software, firmware, and hardware that enforces a security policy. For more detail, see Chapter 2, “*What Is the UNIX System?*“ chapter.
tty	Historically, the abbreviation for a Teletype® terminal. Today, it is generally used to denote any user terminal.
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.
user	Anyone who uses a computer or an operating system.
user-defined	Something determined by the user.
user-defined variable	A named variable given a value by the user. See “variable.”
utility	Software used to carry out routine functions or to assist a programmer or system user in establishing routine tasks.
variable	A symbol whose value may change. In the shell, a variable is a symbol representing some string of characters. 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.”

Index

Spine for 2” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up

PowerMAX OS

User/Administrator

User’s Guide

0890428