SCO® XENIX® System V

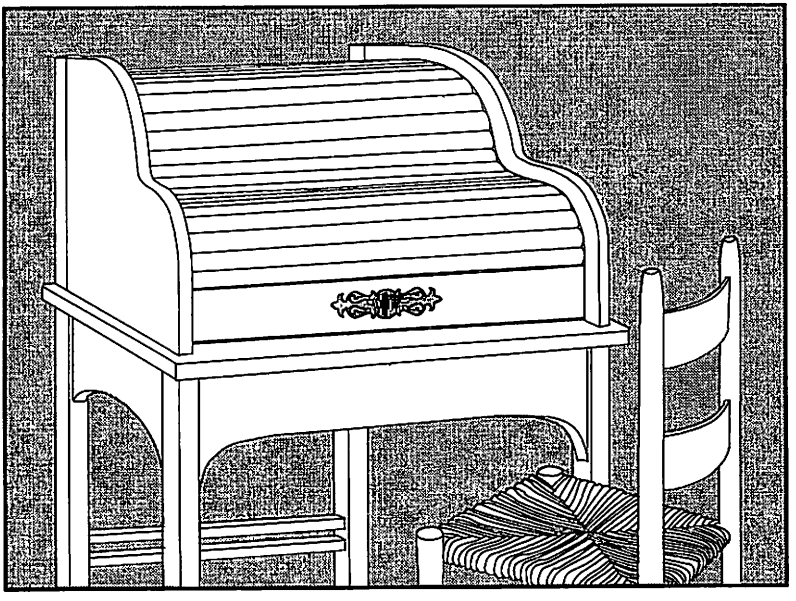
Operating System

Tutorial

The Santa Cruz Operation, Inc.

Document Version 2.3.4C
Date: 28 March 1991

Contents



Chapter 1

Introduction

Introduction

This tutorial is an introduction to the use of your XENIX system. It is intended for users who have little or no familiarity with XENIX systems.

The operating system is a software package that controls the actions of your computer system. It makes it easy for you, as a user, to get the computer to do some very complex tasks.

For example, if you want to find out who is currently using the system, just type the command **who**. The operating system calls up an already-existing program that tells the computer to find out who is logged in, and to display the list of names on your screen. If you had to use a programming language to tell the computer to display the list, it would take several lines of code. With XENIX operating systems, there is no need to learn a programming language, because the programs have already been written for you. You never actually see these programs; what you see are the results of executing them when you type the one-word commands.

Another feature of the XENIX system is that it allows more than one person at a time to use the computer system. It does this by taking advantage of the speed with which computers operate. The operating system stores all of the commands from every user and gives them to the computer's hardware one at a time. The operating system and the hardware work so quickly that each user perceives his or her command as being executed immediately. In fact, you will probably not even be able to tell that anyone else is using the system.

In addition to allowing more than one person to use the system at the same time, XENIX systems also permit the simultaneous running of various printers, other peripherals, and tasks. For these reasons, XENIX systems are referred to as *multi-user, multi-tasking* operating systems.

The aim of this tutorial is to teach you how to do useful work on a XENIX system as quickly as possible. XENIX systems are distributed with over two hundred commands and programs. The commands and programs described in this tutorial are those that you will use most often, and those that you will find most useful. To this end, it is not necessary to provide you with complete information about each command described in this tutorial. For complete information, refer to the appropriate sections of the *XENIX Reference* and the *XENIX User's Guide.*

About This Tutorial

This tutorial is organized as follows:

- Chapter 1, "Introduction," presents an overview of the contents of the entire tutorial, and explains how to use it.

- Chapter 2, "Basic Concepts," explains the concepts that you need to understand to work effectively in the XENIX environment. The chapters that follow presuppose an understanding of the material presented in this chapter.

- Chapter 3, "Logging In," explains how to log in to the system, how to keep your account secure, how to edit the login prompt and how to enter XENIX commands.

- Chapter 4, "Working with Files and Directories," explains how to perform some of the basic tasks involving files and directories. This chapter explains how to create files and directories, how to move, copy, delete, and rename files and directories. The chapter also explains how to use various XENIX text processing utilities, and how to use access permissions with files and directories.

- Chapter 5, "Housekeeping," explains how to use XENIX "housekeeping" utilities. This chapter explains how to create backups, how to copy diskettes, how to get information about the status of the system, and how to place commands in the background. The chapter also contains a brief discussion of shell programming.

- Chapter 6, "XENIX Desktop Utilities," explains how to use the XENIX desktop utilities. This chapter explains how to use the automatic reminder service, how to communicate with other users on the system and how to use the system calculator.

The best way to use this tutorial is to begin by reading Chapter 2. This will provide you with the background information that you need in order to understand the material presented in subsequent chapters. You should then read Chapters 3 through 6 at your terminal, entering commands as instructed in the examples.

Each section of each chapter is a self-contained unit. You do not have to read previous sections in order to understand the material presented in any particular section. If you only need to know how to perform a specific task, you can turn to the section of the chapter that explains how to perform that task. For example, if you already know how to create files but are not sure how to print them, turn to "Printing Files" in Chapter 4, "Working with Files and Directories." In this case, you do not have to read the first sections of Chapter 4 in order to understand "Printing Files."

Notational Conventions

This tutorial uses the following notational conventions:

- Examples in the text are indented.
- *Directories* and *filenames* are printed in *italics.*
- *New concepts* reviewed in the chapter summaries are printed in *italics.*
- **Commands** that you enter are printed in **boldface** type.
- **Keys** to be pressed are printed in **boldface** type. For example, the Return key is represented by:

  ⟨Return⟩

- **Key combinations** are printed in **boldface** and are hyphenated. An example is:

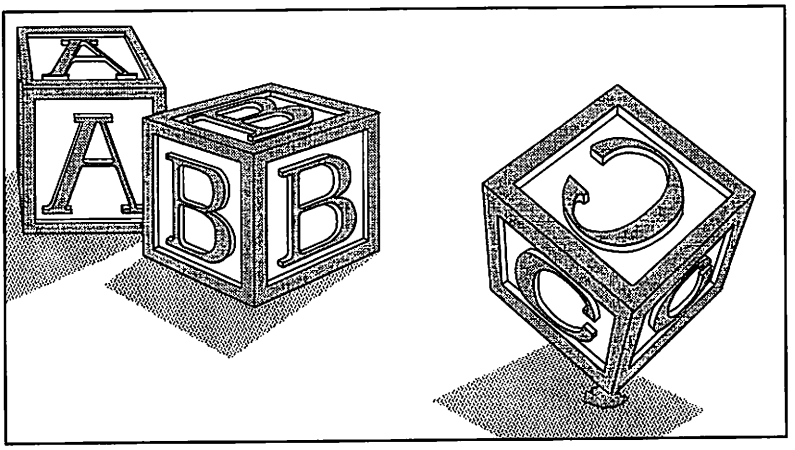
  ⟨Ctrl⟩d

  When you see a key combination, you are supposed to hold down the first key and press the second key. In this example, you should hold down the Control key and press the d key.

- An uppercase letter in parentheses is often appended to command names, as in:

  touch(C)

  The letter in parentheses refers to the section of the *XENIX Reference* that contains complete information on the command.



Chapter 2

Basic Concepts

Introduction

This chapter explains the basic concepts that you need to understand to work effectively in the XENIX environment. After reading this chapter, you should understand the fundamentals of user accounts, as well as how the system's files and directories are organized and named, how commands are entered, and how a command's input and output can be redirected. It is important to read this chapter before proceeding to the tutorial chapters that follow.

Accounts

To organize and record each user's activities, the system administrator gives everyone an account. There are two main types of accounts: User and Super User. Both of these are described in the following sections.

User Accounts

User accounts are the type most commonly issued. They are given to anyone who needs to log in to a XENIX system. Your user account contains the following information:

- Your login name. This is the name by which you are known on the system. It is the name you enter at the login prompt.

- Your password. To increase system security, each user may be given a password. This password is entered when you log in to the system.

- Your group identification. Each user is known to the system as an individual and as a member of a group. Group membership is important for system security reasons. As a member of a group, you may be permitted to access files and directories that you cannot access as an individual.

- Your "home directory." This is the place in the filesystem where you can keep personal files. When you first log in to the system, you are placed in your home directory.

- Your "login shell." This is the program that reads and executes the XENIX commands you input. In most cases, your login shell will be the "Bourne shell." The Bourne shell uses the dollar sign ($) as a prompt. However, you may be configured to use the "C-shell," which uses the percent sign (%) as a prompt. The Korn shell is similar to the Bourne shell, but has advanced features such as command-line editing. The "Visual shell," a menu-driven interface, is also available. Throughout this tutorial, the expression "XENIX prompt" is used to refer to your shell prompt, whether it is the percent sign or the dollar sign.

Once an account has been established for you, you can manipulate the files, directories, and commands that make up a XENIX system.

Super User Account

In addition to each user's individual account, every XENIX system has a "super user" account. (The super user is also referred to as "root.") In order to perform certain system administration tasks, the system administrator must log in as the super user. The super user has free rein over the system. The super user can read and edit any file on the system, as well as execute any program.

Files

The file is the fundamental unit of the XENIX filesystem. There are three different types of XENIX files: ordinary files (what we usually mean when we say "file"), special device files, and directories. Each of these is described in the sections that follow.

Ordinary Files

An ordinary file is simply a collection of 8-bit bytes. Ordinary files are usually documents, program source code, or program data. Executable binary files, or computer programs, are also considered ordinary files. The bytes of an ordinary file are interpreted as text characters, binary instructions, or program statements, by the programs that examine them.

Every ordinary file has the following attributes:

- a filename (not necessarily unique),
- a unique filesystem number called an inode number,
- a size in bytes,
- a time of last change,
- a set of access permissions,
- an owner and a group.

File Protection

On a multi-user system, it is often necessary to "protect" certain files, denying some users access to the files while allowing access to others. Files are protected by assigning appropriate "access permissions" to them. XENIX systems provide three levels of access permissions:

read Having read permission on a file allows a user to view the contents of the file with such commands as **cat** and **more**. A user with read-only permission cannot edit a file.

write Having write permission on a file allows a user to edit the file.

execute If the file is a program, having execute permission on the file allows a user to run the program. You cannot run a program for which you do not have execute permission.

Access permissions are assigned by a file's owner. (By default, the owner of a file is its creator.) Any combination of the three levels is permitted. This allows the file's owner to determine which users can read, write, and/or execute the file. Note that the super user has read, write, and execute permissions on all files on the system.

The XENIX file security mechanism is very flexible. It allows separate access permissions to be set for a file's owner, a file's group, and for all other users. In a typical case, the owner of a file might have read and write permissions, the group read-only permission, and all other users no access permissions at all.

Special Device Files

Each physical device on the system, such as hard and floppy disks, lineprinters, terminals, and system memory, is assigned to a "special file." These files are also called "special device files." Special device files are not discussed in this tutorial. (For more information on special device files, see the *XENIX System Administrator's Guide*.)

Directory Files

Directory files are more like file drawers than files. They are places where files are stored (conceptually, not physically). A directory file is usually referred to as a "directory," and contains the names and locations of the files "within it."

Like ordinary files, directories can be protected by assigning appropriate access permissions. These are read, write and execute. In order to do anything useful in a directory, a user must have execute permission on that directory. Execute and write permissions determine whether files can be added to or removed from a directory. Execute and read permissions determine whether the contents of a directory can be listed. Access permissions are assigned to a directory by its owner. By default, the owner of a directory is its creator.

Directory Structure

With multiple users working on multiple projects, the number of files in a filesystem can proliferate rapidly, creating an organizational nightmare. The inverted "tree-structured" directory hierarchy that is a feature of XENIX systems allows users to organize large numbers of files efficiently. Related files can be grouped together in a single directory. In addition to ordinary files, a directory can contain other directories, sometimes called "subdirectories." Subdirectories themselves can contain ordinary files and more subdirectories, and so on. The **cd** command is used to move from one directory to another.

In this typical tree of files, the root of the tree is at the top and the branches of the tree grow downward. Directories correspond to "nodes" in the tree, while ordinary files correspond to "leaves." Figure 2-1 represents this inverted tree-structured directory hierarchy.

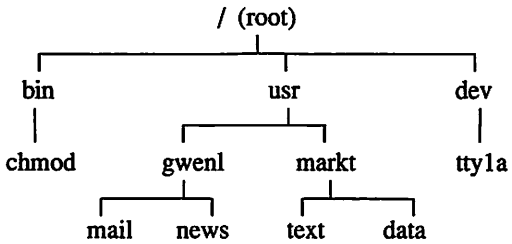


Figure 2-1 A Typical Filesystem

In Figure 2-1, the names *bin*, *usr*, *dev*, *gwenl*, and *markt* all represent directories, and are all nodes in the tree. At the top of the tree is the root directory, which is given the name slash (/). The names *mail*, *news*, *text*, and *data* all represent ordinary files, and they are all "leaves" of the tree. The file *chmod* is the name of a command that can be executed. The name *tty1a* is a special device file. It represents a terminal and is also represented in the tree.

If a directory contains a downward branch to other files or directories, those files and directories are "contained" in the given directory. All directories and files on the system are contained in the root directory. In Figure 2-1, the files *mail* and *news* are contained in the directory *gwenl*, which itself is contained in the directory *usr*. The directory *usr*, in turn, is contained in the root directory.

It is possible to name any file in the system by starting at the root and traveling down any of the branches to the desired file. Files can also be named relative to any directory. XENIX naming conventions are discussed later in this chapter.

The User Directory

Each user is given a personal or "home" directory. This is a place where you can keep files that no other user is likely to need. Within the home directory, you may have other subdirectories that you own and control. All of the home directories on a XENIX system are often placed in the *usr* directory, as illustrated by Figure 2-2.

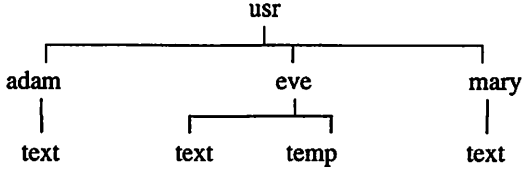


Figure 2-2 A Typical User Directory

In Figure 2-2, the *usr* directory contains each user's home directory. There are three users on this system, *adam*, *eve*, and *mary*.

Naming Conventions

Every single file, directory, and device on a XENIX system has both a filename and a pathname. Filenames and pathnames are discussed in the following two sections.

Filenames

A filename is a sequence of 1 to 14 characters consisting of letters, digits and other special characters such as the underbar (_). Every single file, directory, and device in the system has a filename. Although you can use almost any character in a filename, it is best to confine filenames to the alphanumeric characters and the period. Other characters, especially control characters, are discouraged for use in filenames.

Filenames should be indicative of a file's contents. For example, a file containing purchase orders should have a name like *orders*, rather than *file1*. Note that filenames must be unique only within directories and need not be unique system-wide. Different directories can contain different files that have the same name. For example, there can be several files named *text* on a single system, as long as those files are each in separate directories.

When a filename contains an initial period, it is "hidden," and it is not displayed by the **lc** command. System configuration files are often hidden. However the **lc -a** command does display hidden files. The dash (-) is used in specifying command options and should be avoided when naming files. In addition, the question mark (?), the asterisk (*), brackets ([and]), and all quotation marks should *never* be used in filenames, because they have special meaning to the XENIX shell. (For more information on these characters, see "Special Characters" later in this chapter.)

Pathnames

A pathname is a sequence of directory names followed by a simple filename, each separated from the previous name by a slash. If a pathname begins with a slash, it specifies a file that can be found by beginning a search at the *root* of the entire tree. Otherwise, files are found by beginning the search at the user's *current directory* (also known as the *working directory*). The **pwd** command is used to print the name of the working directory on the screen.

A pathname beginning with a slash is called a *full* or *absolute pathname*. The absolute pathname is a map of a file's location in the system. Absolute pathnames are unique: no two files, directories, or devices have the exact same absolute pathname. A pathname *not* beginning with a slash is called a *relative pathname*, because it specifies a path relative to the current directory.

Sample Names

Among the directory and file names commonly found on XENIX systems are:

/	The name of the root directory.
/bin	The directory containing most of the frequently used XENIX commands.
/usr	The directory containing each user's personal directory. The subdirectory, */usr/bin* contains frequently used XENIX commands not in */bin*.
/dev	The directory containing special device files.
/dev/console	The special device file associated with the system console.
/dev/tty*XX*	The names of special device files associated with system ports. *XX* represents a number, such as *1a* or *006*. Most ports are assigned to terminals.
/lib	The directory containing files of "libraries" used for system development.
/usr/lib	The directory containing directories with XENIX applications.
/tmp	The directory for temporary files.
/usr/joe/run	A typical full pathname. It is the pathname of a file named *run* belonging to a user named *joe*.

bin/script	A relative pathname. It names the file *script* in subdirectory *bin* of the current working directory. If the current directory is the root directory (/), it names */bin/script*. If the current directory is */usr/joe*, it names */usr/joe/bin/script*.
file1	Name of an ordinary file in the current directory.

All files and directories, with the exception of the root directory, have a "parent" directory. This directory is located immediately above the given file or directory. The XENIX filesystem provides special shorthand notations for the parent directory and for the current directory:

. The shorthand name of the current directory. For example, *./filexxx* names the same file as *filexxx*, in the current directory.

.. The shorthand name of the current directory's parent directory. For example, the shorthand name ../.. refers to the directory that is two levels "above" the current directory.

Special Characters

XENIX systems include a facility for specifying sets of filenames that match particular patterns. Suppose, for example, you are working on a large book. The different chapters of the book might be kept in separate files, whose names might be *chpt1*, *chpt2*, *chpt3*, and so on. You might even break each chapter into separate files. For example, you might have files named *chpt1.1*, *chpt1.2*, *chpt1.3*, and so on.

If you want to print the whole book on the lineprinter, you could enter the following command:

lp chap1.1 chap1.2 chap1.3 ...

Entering so many filenames in a command quickly becomes tedious, and will probably lead to mistakes. Fortunately, there is a shortcut. A sequence of names containing a common pattern can be specified with the use of special "wildcard" characters. The wildcard characters discussed in this chapter are:

* Matches zero or more characters

[] Matches any character inside the brackets

? Matches any single character

For example, you can enter:

lp chap*

The asterisk (*) means "zero or more characters of any type," so this command translates into "send all files whose names begin with the word *chap* to the lineprinter." This is a quick and effective way of printing all the files that make up your book.

This shorthand notation is not a unique property of the **lp** command. It can be used with any command. For example, you can list the names of the files in the book by typing:

lc chap*

The asterisk is not limited to the last position in a filename. It can be used anywhere in a filename and can occur several times. An asterisk by itself matches all filenames not containing slashes or beginning with periods:

cat *

This command displays all files in the current directory on your terminal screen.

The asterisk is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4, and 9. You can enter:

lp chap[12349]*

The brackets ([and]) mean "match any of the characters inside the brackets." A range of consecutive letters or digits can be abbreviated, so you can also do this with the following command:

lp chap[1-49]*

(Note that this does *not* try to match *chap1** through *chap49**, but rather *chap1** through *chap4** and *chap9**.) Letters can also be used within brackets: "[a-z]" matches any character in the range "a" through "z".

The question mark (?) matches any single character:

lc ?

This command lists all files that have single-character names. The following command lists information about the first file of each chapter (i.e., *chap1.1*, *chap2.1*, ...):

```
l chap?.1
```

If you need to turn off the special meaning of any of the wildcard characters (*, ?, and [...]) enclose the entire argument in single quotation marks. For example, the following command lists only a file named "?" rather than all one-character filenames:

```
lc '?'
```

Pattern-matching features are discussed further in "The Shell" chapter of the *XENIX User's Guide*.

Commands

You have already been introduced to three useful XENIX commands, **lc**, **lp**, and **cat**. The **lc** command is used to display directory contents, the **lp** command to print files and the **cat** command to display file contents.

Commands are executable programs. When you enter the name of a command, the system looks for a program with that name and executes the program, if it can be found. Command lines can also contain arguments that specify options or files that the program needs. The command line and command syntax are discussed in the next two sections.

Command Line

The XENIX system always reads commands from the "command line." The command line is a line of characters that is read by the shell to determine what actions to perform. (There are four shells available: the Bourne shell, the C-shell, the Korn shell, and the Visual shell.) The shell reads the names of commands from the command line, finds the executable program corresponding to the name of the command, then executes that program. When the program finishes executing, the shell resumes reading the command line.

When you enter commands at a terminal, you are actually editing a line of text called the "command-line buffer." The command-line buffer becomes a command line only when you press ⟨Return⟩. The command-line buffer can be edited with the ⟨Bksp⟩ and ⟨Ctrl⟩u keys. If the **INTERRUPT** key is pressed before ⟨Return⟩, the command-line buffer is erased. (On most keyboards, the ⟨Del⟩ key is the **INTERRUPT** key.)

Multiple commands can be entered on a single command line, provided they are separated by a semicolon (;). For example, the following command line prints out the current date and the name of the current working directory:

```
date; pwd
```

Commands can be submitted for processing in the "background" by appending an ampersand (&) to the command line. This mode of execution is similar to "batch" processing on other systems. The main advantage of placing commands in the background is that you can execute other commands from your terminal in the "foreground" while the background commands execute. For example, the following command outputs disk usage statistics in the directory */usr*, a fairly time-consuming operation, without tying up your terminal:

du /usr > diskuse &

The output of this **du** command is placed in the file *diskuse*, by redirecting output with the greater-than symbol (>). (Redirection of input and output is discussed in "Input and Output" below. Background processing is discussed in "Advanced Tasks.")

Syntax

The general syntax for commands is:

cmd [*options*][*arguments*] [*filename*] [...]

By convention, command names are lowercase. Options are always preceded by a dash (-) and are not required. They are used to modify the command. For example, the **lc** command lists, in columnar format, the contents of a directory. The same command with the **-l** option (**lc -l**) produces a long listing of a directory's contents, including file size, permissions, ownership and date.

In some cases, options can be grouped to form a single option argument, as in the following command:

lc -rl

This command is really a combination of two options, where the **-rl** option selects the option that lists all files in the directory in both reverse alphabetical order and with the long format.

Sometimes multiple options must be given separately, as in the following command:

copy -a -v *source destination*

Here the **-a** option tells the **copy** command to ask the user for confirmation before copying *source* to *destination.* The **-v** option specifies "verbose", which causes **copy** to list the names of the files that are copied, as they are copied.

Other arguments, such as search strings, can also be given, as in the following command:

grep *'string of text' data.file*

The *string of text* in this example is a single argument, and is the series of characters, or string, for which the **grep** command searches in the file *data.file.*

Input and Output

By default, the operating system assumes that input comes from the terminal keyboard and that output goes to the terminal screen. To illustrate typical command input and output, enter:

cat

This command now expects input from your keyboard. It accepts as many lines of input as you enter, until you press ⟨Ctrl⟩d which is the "end-of-file" or "end-of-transmission" indicator.

For example, enter:

this is two lines ⟨Return⟩
of input ⟨Return⟩
⟨Ctrl⟩d

The **cat** command immediately outputs each line as you enter it. Since output is sent to the terminal screen by default, that is where the lines are sent. Thus, the complete session will look like this on your terminal screen:

```
$ cat
this is two lines
this is two lines
of input
of input
$
```

The flow of command input and output can be "redirected" so that input comes from a file instead of from the terminal keyboard and output goes to a file or lineprinter, instead of to the terminal screen. In addition, you can create "pipes" to use the output of one command as the input of another. (Redirection and pipes are discussed in the next two subsections.)

Redirection

On XENIX systems, a file can replace the terminal for either input or output. For example, the following command displays a list of files on your terminal screen:

lc

But if you enter the following command, a list of your files is placed in the file *filelist* (which is created if it does not already exist), rather than sent to the screen:

lc > filelist

The symbol for output redirection, the greater-than sign (>), means "put the output from the command into the following file, rather than display it on the terminal screen." The following command is another way of using the output redirection mechanism:

cat f1 f2 f3 > temp

This command places copies of several files in the file *temp* by redirecting the output of **cat** to that file.

The output append symbol (>>) works very much like the output redirection symbol, except that it means "add to the end of." The following command means "concatenate *file1*, *file2*, and *file3* to the end of whatever is already in *temp*, instead of overwriting and destroying the existing contents."

cat file1 file2 file3 >> temp

As with normal output redirection, if *temp* doesn't already exist, it is created for you.

In a similar way, the input redirection symbol (<) means "take the input for a program from the following file, instead of from the terminal." As an example, you could enter the following command to send a file named *letter.txt* to several people using the XENIX **mail** facility:

mail adam eve mary joe < letter.txt

(See Chapter 6 of this tutorial for information on **mail**.)

Pipes

One of the major innovations of XENIX systems is the concept of a "pipe." A pipe is simply a way to use the output of one command as the input of another, so that the two run as a sequence of commands called a "pipeline."

For example, suppose that you want to find all unique lines in *frank.txt*, *george.txt*, and *hank.txt* and view the result. You could enter the following sequence of commands:

```
sort frank.txt george.txt hank.txt > temp1
uniq < temp1 > temp2
more temp2
rm temp1 temp2
```

But this is more work than is necessary. What you want is to take the output of **sort** and connect it to the input of **uniq**, then take the output of **uniq** and connect it to **more.** You would use the following pipe:

```
sort frank.txt george.txt hank.txt | uniq | more
```

The vertical bar character (|) is used between the **sort** and **uniq** commands to indicate that the output from **sort**, which would normally have been sent to the terminal, is to be used as the input of the **uniq** command, which in turn sends its output to the **more** command for viewing.

The following command is another example of a pipe. The **wc** command counts the number of lines, words, and characters in its input. The **who** command prints a list of users currently logged on, one per line. Thus, the following pipeline tells you the number of users who are logged in by counting the number of lines that come from the **who** command:

```
who | wc -l
```

Notice the difference in output between **wc -l** and **wc**. By default, **wc** tells you how many lines, words, and characters there are in the input. However, **wc -l** tells you only how many lines.

Any program that accepts input from the keyboard can accept input from a pipe instead. Any program that displays output to the terminal screen can send input to a pipe. You can have as many elements in a pipeline as you wish.

Summary

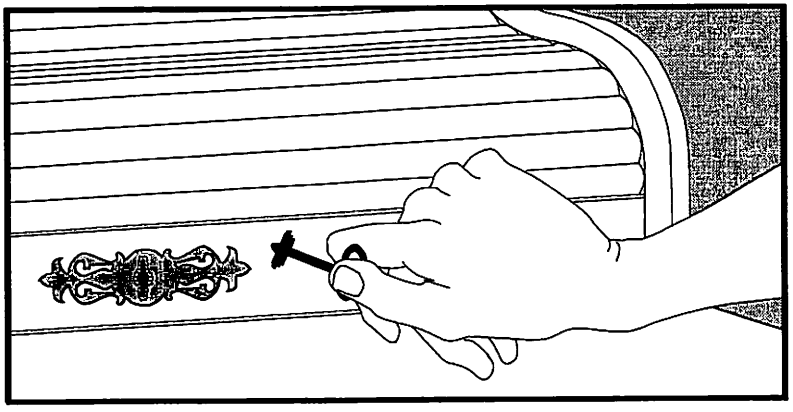
Accounts are assigned to each user to help organize the computer system and to keep track of everyone's activities. Without an account, you cannot log in to the system. There are two types of accounts: *user* and *super user*. User accounts are the more common type, and are given to every user. Super users accounts provide access to all other accounts and files, and are usually only given to the system administrator.

With XENIX systems, all information is stored in *files*. *Special device files* store information about the different hardware components of the system. These pre-made files come with the system, and you cannot manipulate or rename them. *Ordinary files*, on the other hand, can be created, named, and edited by you. Groupings of files are stored in *directories*. Directories can also contain other directories (called *subdirectories*) in addition to files.

You can tell the computer to execute a task by giving it a XENIX *command*. The line on which the command is typed is called the *command line*, and is read by the operating system whenever you press ⟨Return⟩.

You can instruct the operating system to send output to a device other than a terminal screen (such as a printer or a file). Likewise, you can designate an input source to be something other than a terminal. For example, you can use the *pipe* character to tell the operating system to use the output from one command as the input for another. By stringing XENIX commands together in this way, you can create your own customized command sequences.

For a complete explanation of the commands presented in this chapter, see the *XENIX User's Guide* and the *XENIX Reference*.



Chapter 3

Logging In

Introduction

This chapter explains how to perform the following basic tasks on a XENIX system:

- Log in to the system,
- Log out of the system,
- Change your password,
- Reset your terminal type,
- Enter a XENIX command,
- Erase an incorrect command line,
- Stop and start screen output.

This chapter is designed as a tutorial. The best way to use this chapter is to read it at your terminal, entering commands as instructed in the examples.

None of the commands described in this chapter is described in great detail. For a complete explanation of each command, refer to the *XENIX Reference*.

Gaining Access to the System

To use a XENIX system, you must first gain access to it by logging in. When you log in, you are placed in your home directory. Logging in, changing your password, and logging out are described below.

Logging In

Before you can log in to the system, you must be given a system "account." In most cases, your account is created for you by your system administrator. However, if you need to create the account yourself, refer to the *XENIX System Administrator's Guide* for information on creating user accounts. This section assumes that your account has already been created.

Normally, the system sits idle and the prompt "login:" appears on the terminal screen. If your screen is blank or displays nonsense characters, press the ⟨Return⟩ key a few times.

When the "login:" prompt appears, follow these steps:

1. Enter your login name and press ⟨Return⟩. If you make a mistake as you type, press ⟨Ctrl⟩u (hold down the ⟨Ctrl⟩ key and press the **u** key) to start the line again. After you press ⟨Return⟩, "Password:" appears on your screen.

2. Enter your password and press ⟨Return⟩. The letters of your password do not appear on the screen as you enter them, and the cursor does not move. This is to prevent other users from learning your password. If you enter your login name or password incorrectly, the system displays the following message:

```
Login incorrect
login:
```

   If you get this message, enter your login name and password again.

3. Depending on how your system is configured, you may or may not be prompted to enter your terminal type. If you are prompted for your terminal type, you see a line like the following:

```
TERM=(unknown)
```

Enter your terminal type if you see this line. (If you are not sure how to specify your terminal type, contact your system administrator.)

Once you have entered all the correct information, the "prompt character" appears on the screen. This is a dollar sign ($) for Bourne or Korn shell users and a percent sign (%) for C-shell users. The prompt tells you that your XENIX system is ready to accept commands from the keyboard.

Depending on how your system is configured, you may also see a "message of the day" after you log in.

Logging Out

The simplest way to log out is to enter **logout** at the % prompt for C-shell users, or **exit** at the $ prompt for Bourne shell users. You might also be able to logout by pressing ⟨Ctrl⟩d at the prompt. However, some systems are configured to prevent logout with ⟨Ctrl⟩d. The reason for this is that ⟨Ctrl⟩d signifies the end-of-file on XENIX systems, and it is often used within programs to signal the end of input from the keyboard. Since people sometimes make the mistake of pressing ⟨Ctrl⟩d several times, they often find themselves unintentionally logged out of the system. To prevent this, system administrators may disable logout with ⟨Ctrl⟩d.

Familiarize yourself with the logout procedure by pressing ⟨Ctrl⟩d, if you are currently logged in. If this does not work, log out by entering **logout** (C-shell) or **exit** (Bourne or Korn shell). If you are not logged in, log in and then log out, experimenting with ⟨Ctrl⟩d and with **logout** or **exit**.

Changing Your Password

To prevent unauthorized users from gaining access to the system, each authorized user can be given a password. When you are first given an account on a XENIX system, you are assigned a password by the system administrator. Depending on the security scheme used at your site, you may always be assigned passwords, or, allowed to choose your own. Some XENIX systems require you to change your password at regular

intervals. Whether yours does or not, it is a good idea to change your password regularly (at least once every two months) to maintain system security.

Use the **passwd** command to change your password. Follow these steps:

1. Enter the following command and press ⟨Return⟩:

   passwd

   You see:

```
Changing password for user
Old password:
```

   Your login name appears in place of *user*.

2. Carefully enter your old password. It is not displayed on the screen. If you make a mistake, press ⟨Return⟩. The message "Sorry" appears, then the system prompt. Begin again with step 1.

3. The following message appears after you enter your old password and press ⟨Return⟩:

```
Enter new password (minimum 5 characters)
Please use a combination of upper and lower case letters and numbers.
New password:
```

   Enter your new password and press ⟨Return⟩. It is generally a good idea to use a combination of numbers and lower-case and upper-case letters in your password.

4. You see the following message:

```
Re-enter new password:
```

Enter your new password again. If you make a mistake, you see the following message:

```
They don't match; try again
```

Begin again with step 1 if you see this message.

When you complete the procedure, the XENIX prompt reappears. The next time you log in, you must enter your new password.

Keeping Your Account Secure

Security is ultimately the responsibility of the user. The careless use and maintenance of passwords represents the greatest threat to the security of a computer system.

Password Security

Here are some specific guidelines for passwords:

1. Don't use passwords that are easy to guess. Passwords should be at least six characters long and include letters, digits, and punctuation marks. (Example: frAiJ6*)

2. Passwords should not be names (even nicknames), proper nouns, or any word found in */usr/dict/words*. (Don't use a password like: terry9)

3. Always keep your password secret. Passwords should never be written down, sent over electronic mail, or verbally communicated. (Treat it like the PIN number for your instant teller card.)

Good Security Habits

There are simple, good security habits. Here are some general guidelines:

1. Remember to log out before leaving a terminal.

2. Use the **lock**(C) utility when you leave your terminal, even for a short time.

3. Make certain that sensitive files are not publicly readable. (See the discussion of file and directory permissions in Chapter 4 of this tutorial for information on how to do this.)

4. Keep any floppies or tapes containing confidential data (program source, database backups) under lock and key.

5. If you notice strange files in your directories, or find other evidence that your account has been tampered with, tell your system administrator.

Changing Your Terminal Type

To communicate with the operating system via your terminal, you must tell it what type of terminal you have. On most systems, the system console is already configured for use. However, serial terminals of various types can be connected to a XENIX system. If you are working from a serial terminal, it can be important to know how to specify your terminal type.

The terminal type is displayed each time you log in. You can change the value of the terminal type displayed by editing the *.profile* file in your home directory. If you are using the C-shell, you do not have a *.profile* file. Instead, you must edit the *.login* file in your home directory.

There are at least two reasons why you might want to change the value of the terminal type displayed:

- You might have a new terminal that is not the same model as your old terminal. If so, the terminal type displayed by your old *.profile* (*.login*) file will be incorrect.

- The terminal type displayed might be "unknown" or "ansi" or another setting which is not correct for your terminal. This would require you to type in your terminal type every time you log in. By changing the terminal type to the setting that is correct for your terminal, all you have to do is press ⟨Return⟩ when prompted for your terminal type. There is no need for you to enter the terminal type.

To permanently change the terminal type displayed, use **vi** to edit *.profile* (*.login*). In order to use **vi** to make these changes, it may be necessary to manually set the terminal type for the current session. To make this initial specification, enter the commands listed below.

Bourne or Korn shell:

TERM=*termtype*; **export TERM**

C-shell:

setenv TERM *termtype*

where *termtype* is your terminal type. The **terminals**(M) manual page contains a list of supported terminals. In addition, the file */etc/termcap* contains entries for all terminals supported under XENIX.

Once you have temporarily set the terminal type, you can use **vi** to make the appropriate changes in *.profile* (*.login*) so that the the terminal type is set automatically whenever you log in. Chapter 4 of this tutorial explains how to use **vi**.

Once in **vi**, move the cursor to the line that looks like the following:

```
eval 'tset -m :\?unknown -s -r -Q'
```

Change *unknown* (or whatever the value is) in this line to the terminal type of your terminal. For example, if you normally log in on a vt100 terminal, you would change the line to:

```
eval 'tset -m :\?vt100 -s -r -Q'
```

Each time you log in, you would then see the message:

```
TERM = (vt100)
```

Press ⟨Return⟩ and the terminal type is set to vt100. There is no need to enter **vt100**.

Entering Commands

Before you begin working with the commands described in the rest of this tutorial, you should be familiar with three very useful XENIX features. These are character type-ahead and the special key-combinations used to erase the command line, and stop and start screen output. These features are discussed below.

Entering a Command Line

Entering a command line consists of typing characters and then pressing ⟨Return⟩. Once you press ⟨Return⟩, the computer reads the command line and executes the specified commands. No command entered on the command line is executed until ⟨Return⟩ is pressed.

You can enter as many command lines as you want without waiting for the commands to complete their execution and for the prompt to reappear. This is because XENIX systems support character type-ahead. The system can hold up to 256 characters in the kernel buffers that read keyboard input. Experiment with this type-ahead feature by entering the following commands, one right after the other, without waiting for a previous command to finish executing. (Always press ⟨Return⟩ after entering a command. In the following example, press ⟨Return⟩ after entering each command.)

```
lc -la
du -a
lc -Fa
```

These commands generate a long listing of all the files in the current directory, then display disk usage statistics for these files, and finally list the files again, but in a different format.

Erasing a Command Line

Typing errors are bound to occur when you enter commands. To erase the current command line, press ⟨Ctrl⟩u. When you press ⟨Ctrl⟩u, the command line is ignored and the cursor skips to the next line. Press ⟨Return⟩ to get the prompt back.

Halting Screen Output

Data often scrolls across your screen faster than you can read it. To halt scrolling temporarily, press ⟨Ctrl⟩s. To restart scrolling, press ⟨Ctrl⟩q. Experiment with ⟨Ctrl⟩s and ⟨Ctrl⟩q by entering the following command, then pressing ⟨Ctrl⟩s to stop the output and ⟨Ctrl⟩q to restart it:

ls /bin

Summary

For security reasons, XENIX systems require that all users have a *login name* and a *password*. To access the operating system, enter your login name at the login prompt, and your password at the password prompt. If you enter either incorrectly, you must start over from the beginning.

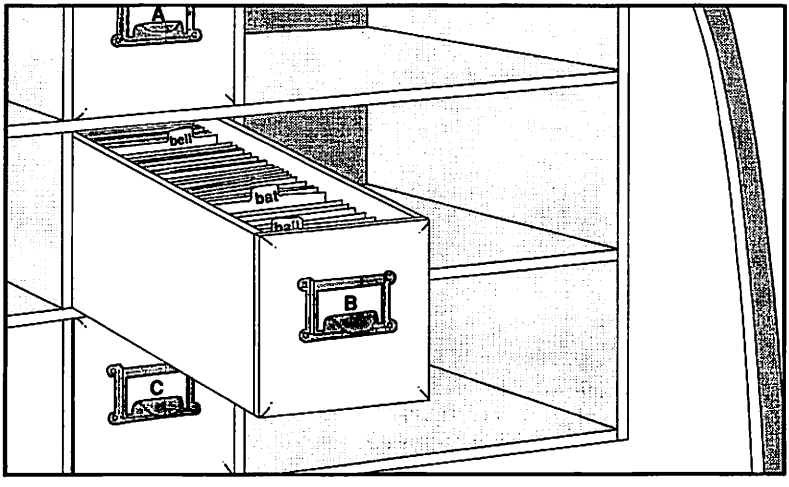
Once you have accessed the system, you can change your password at any time by using the **passwd** command. To enter this or any other XENIX command, you must press ⟨Return⟩ after typing the name of the command on the *command* line. If you need to erase the command line, press ⟨Ctrl⟩u.

The ⟨Ctrl⟩s and ⟨Ctrl⟩q keys respectively stop and continue the scrolling of information across your terminal screen.

If you need to permanently change your terminal type designation, you must edit the file (*.profile* or *.login*) that contains the designation. This file is always in your home directory.

When you are through using the system, enter **logout** if you are using the C-shell, or **exit** if you are using the Bourne or Korn shell. After entering this command, you are logged out of the system.

For a complete explanation of the commands presented in this chapter, see the *XENIX User's Guide* and the *XENIX Reference*.



Chapter 4

Working with Files and Directories

Introduction

This chapter explains how to perform the following tasks on a XENIX system:

- Print the name of the current directory,
- List directory contents,
- Change to another directory,
- Create, remove, rename, and copy directories,
- Display the contents of files,
- Delete, combine, rename, move, copy, and search for files,
- Use the full-screen text editor **vi** to create files,
- Print files,
- Compare and sort files,
- Search for patterns in a file,
- Count words, lines and characters in a file,
- Use file and directory permissions.

This chapter is designed as a tutorial. The best way to use this chapter is to read it at your terminal, entering commands as instructed in the examples.

None of the commands described in this chapter is described in great detail. For a complete explanation of each command, refer to the *XENIX Reference*.

Working with Directories

Because of the hierarchical structure of the XENIX filesystem, any XENIX system has many directories and subdirectories. There are several commands that simplify working in directories. These commands are described in the following sections.

Printing the Name of Your Working Directory

The directory you are "in" at any given time is your "working" directory. All commands are executed relative to the working directory. The name of this directory is given by the **pwd** command, which stands for "print working directory." To find out what your current working directory is, enter the following command:


```
pwd
```

When you first log in to the system, you are placed in your home directory.

Listing Directory Contents

Several related commands are used to list the contents of directories:

lc This command is a variation of the **ls** command. The **ls** command alphabetizes and displays directory contents. The **lc** command alphabetizes directory contents and displays them in columnar format.

lf This command does the same as **lc**, and it also marks directories with a slash (/) and executable files (computer programs) with an asterisk (*).

Enter the following command to list the contents of */usr/bin*:

```
lc /usr/bin
```

This directory contains many of the executable files with which you work in the XENIX environment. Entering **lc** with no directory name lists the contents of the current directory.

The **l** command is also useful. It is equivalent to the **ls -l** command, which produces a "long" listing of a directory's contents. Here is an example output of a long listing:

```
total 338
-rw-rw-r--  1 markt pub  4448 Mar  1 09:16 1.intro.00
-rw-rw-r--  1 markt pub  4457 Mar  1 09:29 1.intro.s
-rw-rw-r--  1 markt pub 33836 Mar  1 09:30 2.concepts.00
-rw-rw-r--  1 markt pub 35096 Mar  1 12:49 2.concepts.s
-rw-rw-r--  1 markt pub 52197 Mar  1 15:09 3.basic.s
-rw-rw-rw-  1 markt pub 39835 Feb 16 11:02 4.advan.s
```

Reading from left to right, the information given for each file or directory by the **l** command includes:

- Permissions - The permissions for the first file in the above figure are **-rw-rw-r--**. This line tells you that the file is an ordinary file, that the owner and group members have read and write permission, and that all other users have read permission. For details about file and directory permissions, see "Using File and Directory Permissions" later in this chapter.
- Number of links - A link is a path to a file. In the above example, all of the files have one link.
- Owner - The owner's name is the login name of the person who created the file.
- Group - A group is an organization of users set up by the system administrator.
- File size in bytes
- Time of last modification
- Filename

The figure at the top lists the total number of "blocks" used on the disk to store these files. A single block is 512 bytes. In the case of the example shown, the files account for 338 blocks, or 173056 bytes.

Changing Your Working Directory

Your working directory represents your location in the filesystem. To move to a new location in the XENIX filesystem, use the **cd** command.

Entering **cd** with no arguments places you in your home directory. Try it. Enter **cd**. To be sure you are now in your home directory, enter **pwd**.

To move to a directory other than your home directory, you must specify that directory as an argument to the **cd** command. For example, enter the following command to move to */usr/bin*:

cd /usr/bin

Verify that you are in */usr/bin* by entering **pwd**.

Change to the "root" directory by entering the following command:

cd /

The root directory is at the "top" of the filesystem. All other directories are "below" it. Enter **lf** to examine the files and directories in the root directory. Then enter **cd** to return to your home directory. (For more information on the root directory, refer to Chapter 2 of this tutorial.)

Some shorthand notation is available to help you move quickly through the filesystem. To move up one directory from your current directory, enter:

cd ..

Enter the following command to move up two directories:

cd ../..

If you entered this latter command from your home directory, you are probably in the root directory. Verify this by entering **pwd**.

Creating Directories

To create a subdirectory in your working directory, use the **mkdir** command. Enter **cd** to move to your home directory and then enter the following command to create a subdirectory named *tempdir*:

mkdir tempdir

Verify that *tempdir* exists with the **lf** command. Change to *tempdir* with the **cd** command and verify that *tempdir* is empty with another **lf** command. Finally, use the **touch**(C) command to create two empty files in *tempdir*:

touch tempfile1 tempfile2

Enter **lf** one more time to verify that *tempfile1* and *tempfile2* were created.

You can only create subdirectories in a directory if you have write permission in that directory. If you do not have write permission and you use **mkdir** to create a subdirectory, you see the following message:

```
mkdir: cannot access directory_name
```

In this message, *directory_name* refers to the directory in which you attempted to create a subdirectory. Verify this by trying to create a subdirectory in the */etc* directory, a directory in which you probably do not have write permission:

mkdir /etc/temp

Removing Directories

Use the **rmdir** command to remove a directory. This command will not work if the directory has files or subdirectories in it. Verify this by moving to your home directory with the **cd** command and then entering the following command to remove *tempdir*, the directory created earlier in "Creating Directories:"

rmdir tempdir

You should see the following message:

```
rmdir: tempdir not empty
```

You must remove *tempfile1* and *tempfile2* from *tempdir* before **rmdir** deletes *tempdir*. But don't remove these files just yet. They are used in another example later in this chapter.

Renaming Directories

To rename a directory, use the **mv** command. For example, **cd** to your home directory and then enter the following command to rename *tempdir*, the directory created earlier in "Creating Directories," to *newdir*:

mv tempdir newdir

Verify the name change by entering **lf**. Note that the files in *newdir* are unaffected by the change. Verify this by entering the following command:

lf newdir

Copying Directories

The **copy** command copies directories. Of course, before you can copy the contents of one directory into another, you must have write permissions on the second directory.

To copy the */newdir* directory created earlier in "Renaming Directories" to */tmp/newdir*, enter the following command:

copy $HOME/newdir /tmp/newdir

In this command, "$HOME" is shorthand for the pathname of your home directory. You can use it wherever you would enter the pathname of your home directory.

When you make a copy of a directory, all or the files in the directory are copied to the new directory. To verify that the files in *$HOME/newdir* were copied to */tmp/newdir*, enter the following command:

lf /tmp/newdir

Remove */tmp/newdir* by entering the following commands:

rm /tmp/newdir/*
rmdir /tmp/newdir

The first command removes the files in */tmp/newdir*, the second command removes */tmp/newdir*. Verify that */tmp/newdir* is removed by entering the following command:

```
lf /tmp
```

Remove *$HOME/newdir* by entering the following commands:

```
rm $HOME/newdir/*
rmdir $HOME/newdir
```

Working with Files

File manipulation (creating, deleting, displaying, combining, renaming, moving, and copying) is one of the most important capabilities an operating system provides. The XENIX commands that perform these functions are described in the following sections.

Displaying File Contents

The **more** command displays the contents of a file, one screenful at a time. It cannot be used to edit files. If the file contains more than one screenful of data, you see the following prompt after each screen of text is displayed:


```
--More--(XX%)
```

XX% represents the percentage of the file displayed. Press the ⟨Return⟩ key to display another line. Press the ⟨Space⟩ (spacebar) to display another screen.

Try the following command:

more /etc/termcap

This causes the contents of */etc/termcap* to display on the screen. To quit **more** before */etc/termcap* is finished displaying, press **q** for quit.

The **more** command does not allow you to scroll backward, toward the beginning of the file. However, you can search forward for patterns with **more** by using the slash (/) command. For example, enter the following commands to search for a line containing "process" in */etc/termcap*:

more /etc/termcap
/process

You see the following message at the top of the screen:

```
...skipping
```

If the pattern is found, it is displayed two lines below this message. If the pattern is not found, "Pattern not found" is displayed.

If you are looking at a file with **more** and decide that you want to edit the file, you can invoke the **vi** editor by pressing **v**. Of course, you must have write permission on a file before you can edit it with **vi** or any other text editor. To display the file's contents, you only need read permission.

You will often use **more** in pipes. For example, **more** is useful when you want to list the contents of a directory in long format. Enter the following command to display a long listing of the contents of */bin*, one screenful at a time:

```
l /bin | more
```

(For more information on pipes, refer to Chapter 2 of this tutorial.)

The **head** and **tail** commands display the beginning and the end of a file, respectively. With no options, they display the first or last 10 lines. Enter the following command to display the last 10 lines of */etc/termcap*:

```
tail /etc/termcap
```


You can specify exactly how many lines you want displayed. Enter the following command to display the first 20 lines of */etc/termcap*:

```
head -20 /etc/termcap
```

Enter the following command to display the last 20 lines of */etc/termcap*:

```
tail -20 /etc/termcap
```

The **cat** command also displays the contents of a file. Unlike **more**, **cat** continuously scrolls the file until you stop the scroll with ⟨Ctrl⟩s. ⟨Ctrl⟩q continues scrolling. Scrolling stops automatically when the end of the file is reached. To stop scrolling before the end of the file, press **INTERRUPT**, which is the ⟨Del⟩ key on most keyboards.

Enter the following command to display the contents of */etc/termcap*. Use ⟨Ctrl⟩s and ⟨Ctrl⟩q to stop and start the scrolling and **INTERRUPT** to halt the scrolling before the end of the file is reached:

```
cat /etc/termcap
```

Listing Invisible (Dot) Files

Filenames beginning with a period (dot) are invisible; the normal listing commands like **l**, **lc**, etc., do not display these files. These commands include a **-a** option that lists **all** files. For example, the command:

lc -a

might display a list of files like this:

```
.       ..      .cshrc  .login  prints   quotes  globals
```

The "." and ".." refer to the present and upper directories, respectively. The files *.cshrc* and *.login* are invisible files.

Deleting Files

The **rm** command is used to delete files. We have used it throughout this chapter to delete various files. Use **cd** to change to your home directory and enter the following command to create three new files:

touch tempfile1 tempfile2 tempfile3

Delete *tempfile3* by entering the following command:

rm tempfile3

The **-i** option allows you to remove files interactively by asking you if you really want to delete each of the files specified on the command line. If you press **y** followed by a ⟨Return⟩, the given file is removed. If you press **n**, the file is left untouched. This option is useful when removing files from a directory that contains many files. It helps you avoid erasing files accidentally that you really want to keep.

Experiment with the **-i** option by entering the following command:

rm -i tempfile1 tempfile2

Note that you can place several filenames on the **rm** command line. This is true for most XENIX commands. You can also use wildcard characters. For example, instead of entering the above command, you could enter the following:

rm -i tempfile*

(The use of wildcard characters on the command line is discussed in Chapter 2 of this tutorial.)

Combining Files

In addition to displaying files, the **cat** command can be used to combine several existing files into a single new file. This is done by redirecting the output of **cat** into a new file. The greater-than sign (>) is used for redirection. If the new file does not exits, it is created automatically. (If you are not familiar with redirection, see Chapter 2 of this tutorial.)

Use **cd** to move to your home directory and enter the following command to combine */etc/motd* and */etc/default/tar* into a file named *catfile*:

cat /etc/motd /etc/default/tar > catfile

Now display the contents of the new file *catfile* with the **more** command:

more catfile

The symbol >> can be used with **cat** to *append* one file to the end of another file. For example, to append the contents of */etc/motd* to *catfile*, enter the following command:

cat /etc/motd >> catfile

The contents of */etc/motd* should now be placed at the beginning and at the end of *catfile*. Verify this with the following *head* and *tail* commands:

head -20 catfile
tail -20 catfile

Renaming Files

The **mv** command is used to move files around the XENIX filesystem and also to rename files. Use **cd** to move to your home directory. Rename *catfile*, created earlier in "Combining Files,"to *catfile2* by entering the following command:

mv catfile catfile2

After this move is completed, *catfile* no longer exists. The file *catfile2* exists in its place. Verify this by entering the following command:

lc

Moving Files

To move a file into another directory, give the name of the destination directory as the final name in the **mv** command. You do not need to specify the destination filename. For example, enter the following command to move *catfile2*, created earlier in "Renaming Files," to the */tmp* directory:

mv $HOME/catfile2 /tmp

To be sure that *catfile2* is in */tmp* and not in the current directory, enter the following command:

lc . /tmp

(Remember that you can enter more than one argument on most command lines, and that the dot (**.**) stands for the current directory.)

Finally, move *catfile2* back to the current directory by entering the following command:

mv /tmp/catfile2 .

The **mv** command always checks to see if the last argument is the name of a directory. If it is, all files designated by filename arguments are moved into that directory. However, if you do not have write permission on the directory to which you are attempting to move files, the move fails.

Copying Files

The **cp** command is used to copy files. There are two forms of the **cp** command, one in which files are copied into a directory and another in which a file is copied to another file.

Use **cd** to change to your home directory. Then enter the following command to copy the contents of *catfile2*, created earlier in "Renaming Files," to *catfile3*:

cp catfile2 catfile3

You now have two files with identical contents. To copy *catfile2* and *catfile3* to the */tmp* directory, enter the following command:

cp catfile2 catfile3 /tmp

This last command can be simplified by using a wildcard character:

cp catfile* /tmp

Like the **mv** command, **cp** always checks to see if the last argument is the name of a directory, and, if so, all files designated by filename arguments are copied into that directory. However, unlike the **mv** command, **cp** leaves the original file untouched. There should now be two copies of *catfile2* and *catfile3* on the system, one copy of each in the current directory and one copy of each in */tmp*.

Finding Files

A XENIX filesystem can contain thousands of files. Because of this, files can often get lost. The **find** command is used to search the filesystem for files. The command has the form:

find *pathname* **-name** *filename* **-print**

The *pathname* is the pathname of the directory that you want to search. The search is recursive; it starts at the directory named and searches downward through all files and subdirectories under the named directory.

The **-name** option indicates that you are searching for files that have a specific *filename*. The **-print** option indicates that you want to print the pathnames of all the files that match *filename* on your screen.

Enter the following command to search all directories and subdirectories for *catfile2*, the file created earlier in "Renaming Files:"

find / -name catfile2 -print

It may take a few minutes for this command to finish executing. The output of this **find** command should indicate that there are at least two occurrences of *catfile2*, one in */tmp* and one in your home directory. Remove *catfile2* and *catfile3* from */tmp* and from your home directory by entering the following command:

rm /tmp/catfile* $HOME/catfile*

Editing Files with vi

The **vi** text editor is a full-screen editor included with XENIX operating systems. The sections that follow briefly explain how to use **vi**. For a more complete discussion, see the *XENIX User's Guide*.

Entering Text

Change to your home directory with the **cd** command and enter the following command to create a file called *tempfile*:

vi tempfile

A message appears indicating that you are creating a new file. You are then placed in **vi**.

There are two modes in **vi**: *Insert mode* and *Command mode*. Use Insert mode to add text to a file. Use Command mode to edit existing text in a file. Since *tempfile* is empty, press **i** to enter Insert mode.

Enter the following lines of text, pressing ⟨Return⟩ after each line. If you make a mistake typing, use the ⟨Bksp⟩ key to erase the mistake and continue typing:

This tutorial is very, very helpful.
It makes learning to use a XENIX system easy.
I'm glad I have this tutorial.

After you enter the last line, press the ⟨Esc⟩ key. It takes you out of Insert mode and places you in Command mode.

Moving the Cursor

Although many cursor-movement commands are available in **vi**, only the four basic ones are discussed here:

h When you are in Command mode, pressing the **h** key moves the cursor one character to the left.

l When you are in Command mode, pressing the **l** key moves the cursor one character to the right.

k When you are in Command mode, pressing the **k** key moves the cursor up one line.

j When you are in Command mode, pressing the **j** key moves the cursor down one line.

Experiment with these cursor-movement keys on the text you entered. Note that, if your keyboard has arrow keys, these usually perform in the manner of **h, l, k** and **j**.

Deleting Text

Deleting text with **vi** is very easy. Different commands allow you to delete characters, words and entire lines.

To delete a single character, place the cursor on that character with the cursor-movement keys and press the **x** key. Experiment with the **x** key by deleting the comma in the first line.

To delete a word, place the cursor on the first character of the word and press **dw** (press **d**, release it, and press **w**). Experiment with this by placing the cursor on the first character of "very" in the first line and pressing **dw**.

To delete an entire line, place the cursor anywhere on that line and press **dd** (press **d**, release it, and press **d** again). Experiment with this by placing the cursor on the third line and pressing **dd**. Your file should now contain the following text:

```
This tutorial is very helpful.
It makes learning to use the system easy.
```

Inserting Text

The **i** and **o** keys can be used to insert text. We have already used the **i** key to enter text in an empty file. To enter additional text on an existing line, move the cursor to the point where you want the new text to begin, press **i** to enter Insert mode, enter the text, and press ⟨Esc⟩ to return to Command mode. For example, move the cursor to the "e" in "easy" in the second line, press **i**, enter the word **very**, press the ⟨Space⟩, and press ⟨Esc⟩ to return to Command mode. The second line should now be:

```
It makes learning to use the system very easy.
```

The **o** key can be used to insert a new line. To use it, move the cursor to the line *directly above* the place in the file where the new line is to be inserted and press **o**. A new line is inserted, with the cursor placed at the beginning. You are also automatically placed in Insert mode. Try this by moving the cursor to the second line of *tempfile* and press **o**. Now enter more text. Press ⟨Esc⟩ when you are finished.

Leaving vi

Most of the time, you will want to save your file before leaving **vi**. To do this, enter Command mode and type **:x**. This command saves the file you are editing and returns you to the XENIX prompt.

In some cases, you will want to leave **vi** without saving your work. To do this, enter Command mode and type **:q!**. This command returns you to the XENIX prompt, without saving the changes that you made to your file.

Leave *tempfile* by pressing **:x**. Re-enter *tempfile* by entering the following command:

vi tempfile

Insert some text using either the **i** or the **o** key, press ⟨Esc⟩ and then enter **q!** to quit without saving your changes. Display *tempfile* by entering the following command:

cat tempfile

You will notice that the last set of changes you made do not appear. Remove *tempfile* by entering the following command:

rm tempfile

Printing Files

To print files, use the **lp** command. This is one of a group of commands known as the print service commands. The lineprinter commands are easy to use and very flexible. With a few simple commands, you can print multiple copies of a file, cancel a print request, or ask for a special option on a particular printer. Check with your system administrator to find out what lineprinters and printer options are available on your system.

Using lp

Use **cd** to change to your home directory and enter the following command to create a file with which you can experiment:

cp /etc/motd $HOME/printfile

This command places a copy of */etc/motd* in your home directory, naming it *printfile*. The file */etc/motd* is the "message of the day file." Its contents are displayed every time you log in to a XENIX system.

A directory must be "publicly executable" before you can use **lp** to print any of the files in that directory. This means that other users must have execute permissions on the directory. Enter the following command to make your home directory publicly executable:

chmod o+x $HOME

(See "Using File and Directory Permissions" later in this chapter for more information on **chmod**(C).)

Enter the following command to print *printfile*:

lp printfile

This command causes one copy of *printfile* to print on the default printer on your system. A banner page might be printed along with the file. Note that you can print several files at once by putting more than one name on the **lp** command line.

When you print with **lp**, a "request ID" is displayed on your screen. A request ID might look like the following:

pr4-532

The first part (pr4) is the name of the printer on which your file is printed. The second part (532) identifies your job number. Should you later wish to cancel your print request or check its status, you will need to remember your request ID. (Canceling and checking on print requests are discussed below.)

You can also use **lp** with pipes. For example, enter the following command to sort and then print a copy of */etc/passwd*, the file that contains system account information:

sort /etc/passwd | lp

(For more information on **sort**(C), see "Sorting Files" later in this chapter.)

Using lp Options

The **lp** command has several options that help you control the printed output. You can specify the number of copies you want printed by using the number option, **-n**. For example, to print two copies of *printfile*, enter:

lp printfile -n2

Several different printers are often attached to a single XENIX system. With the **-d** option, you can specify the printer on which your file is to be printed. To print two copies of *printfile* on a printer named *laser*, enter:

lp *printfile -n2* **-dlaser**

Check with your system administrator for the names of the printers available on your system.

Canceling a Print Request

Use the **cancel** command to cancel any of your print requests. With this command, you can only cancel your own requests. The system administrator is the only person allowed to cancel the requests of other users. If the system administrator cancels one of your print requests, you are automatically notified via mail.

The **cancel** command takes as its argument the request ID. For example, to stop printing one of your files with a request ID of *laser-245*, you would enter:

cancel laser-245

Experiment with **cancel** by using **lp** to print *printfile* and then using **cancel** to cancel the print request. When you are finished, enter the following command to remove *printfile*:

rm printfile

You can also use the **cancel** command to stop whatever is currently printing on a particular printer. For example, to cancel whatever file is currently printing on the printer *laser*, you would enter the following command:

cancel laser

If you cancel a file that does not belong to you, mail reporting that the print request was canceled is automatically sent to the file's owner.

Finding Out the Status of a Print Request

Use the **lpstat** command to check on the status of your print request. To use it, simply enter the following:

lpstat

The **lpstat** command produces output like the following:

```
prt1-121    cindym    450   Dec 15 09:30
laser-450   cindym   4968   Dec 15 09:46
```

Note that entering **lpstat** with no options displays information on your files only, not those of other users. To generate a report for all users on your computer, use **lpstat** with the **-o** option. Nothing is displayed by the **lpstat** command if the print job is complete.

The first column of the **lpstat** output shows the request ID for each of your files being printed. The second column is your login name. In the third column, the number of characters to be printed is shown, and the fourth column lists the dates and times the print requests were made.

To learn the status of a particular file, use the **lpstat** command with the file's request ID. For example, to find out the status of a file with the request ID of *laser-256*, you would enter the following command:

lpstat laser-256

The status of that file only is displayed.

You can also request the status of various printers on your system by using the **-p** option or by giving the name of the particular printer you are interested in. Enter the following command to find out the status of all the printers on your system:

lpstat -p

To find out the status of a printer named *laser*, you would enter the following:

lpstat -plaser

The request ID and status information for each file currently waiting to be printed on *laser* is displayed.

Processing Text Files

XENIX systems include a set of utilities that let you process information in text files. These utilities enable you to compare the contents of two files, sort files, search for patterns in files, and count the characters, words, and lines in files. These utilities are described below.

Comparing Files

The **diff** command allows you to compare the contents of two files and to print out those lines that differ between the files. To experiment with **diff**, use **vi** to create two files to compare. The files will be *men* and *women*. First **cd** to your home directory. Then enter the following command at the XENIX prompt:

vi men

When you are placed in **vi**, press the **i** key to enter Insert mode, and then type the following lines:

Now is the time for all good men to
Come to the aid of their party.

Press ⟨Esc⟩ to return to Command mode and save *men* by entering **:w**. While still in Command mode, enter the following command to create *women*:

:n women

You see the following message:

```
"women" No such file or directory
```

You are then placed in *women*. Press **i** to enter Insert mode and then enter the following lines:

Now is the time for all good women to
Come to the aid of their party.

Press ⟨Esc⟩ to return to Command mode, then **:x** to save *women* and leave **vi**. You have now created *men* and *women*.

Enter the following command to compare the contents of these two files:

diff men women

This **diff** command should produce the following output:

```
1c1
< Now is the time for all good men to
---
> Now is the time for all good women to
```

The lines displayed are the lines that differ from one another in the two files.

Sorting Files

One of the most useful file processing commands is **sort**. When used without options, **sort** alphabetizes lines in a file, starting with the leftmost character of each line. These sorted lines are then output to the screen, or to a file if redirection is used on the **sort** command line. This command does not affect the contents of the actual file.

Enter the following command to display an alphabetized list of all users who have system accounts:

sort /etc/passwd

The **sort** command is useful in pipes. Enter the following command to display an alphabetized list of users who are currently using the system:

who | sort

Searching for Patterns in a File

The **grep** command selects and extracts lines from a file, printing only those lines that match a given pattern. Enter the following command to print out the lines in */etc/passwd* that contain your login information. There will probably be only one such line:

grep *login* /etc/passwd

Be sure to replace *login* in this command with your login name. Your output should be similar to the following:

```
forbin:OVbYnTHxp:6005:104:Dr. Charles Forbin, CPO:/u/forbin:/bin/csh
```

Note that whenever wildcard characters are used to specify a **grep** search pattern, the pattern should be enclosed in single quotation marks (' '). Note also that the search pattern is case sensitive. Searching for "joe" will not yield lines containing "Joe".

As another example, assume that you have a file named *phonelist* that contains a name followed by a phone number on each line. Assume also that there are several thousand lines in this list. You can use **grep** to find the phone number of someone named Joe, whose phone number prefix is 822, by entering the following command:

grep 'Joe' phonelist | grep '822-' > joes.number

The **grep** utility first finds all occurrences of lines containing the word "Joe" in the file *phonelist*. The output from this command is then filtered through another **grep** command, which searches for an "822-" prefix, thus removing any unwanted "Joes." Finally, assuming that a unique phone number for Joe exists with the "822-" prefix, that name and number are placed in the file *joes.number*.

Two other pattern searching utilities are available. These are **egrep** and **fgrep.** Refer to **grep**(C) in the *XENIX Reference* for more information on these utilities.

Counting Words, Lines, and Characters

The **wc** utility is used to count words in a file. Words are presumed to be separated by punctuation, spaces, tabs, or newlines. In addition to counting words, **wc** counts characters and lines.

Use **cd** to change to your home directory. Then enter the following command to count the lines, words, and characters in the file *men*, created earlier in "Comparing Files:"

wc men

The output from this command should be the following:

```
  2     16     68 men
```

The first number is the number of lines in *men*, the second number is the number of words and the third number is the number of characters. Remove *men* and *women* by entering the following command:

rm *men

To specify a count of characters, words, or lines only, you must use the **-c**, **-w**, or **-l** option, respectively. For example, enter the following command to count the number of users currently logged onto the system:

who | wc -l

The **who** command reports on who is using the system, one user per line. The **wc -l** command counts the number of lines reported by the **who** command. This is the number of users currently on the system.

Using File and Directory Permissions

XENIX systems allow the owner of a file or directory to restrict access to that file or directory. This is done with permission settings. Permissions on a file limit who can read, write and/or execute the files. Permissions on a directory limit who can **cd** to the directory, list the contents of the directory, and create and delete files in the directory.

To determine the permissions associated with a given file or directory, use the **l** command. Use **cd** to change to your home directory and then enter **l** to get a long listing of the files in this directory.

Permissions are indicated by the first 10 characters of the output of the **l** command. The first character indicates the type of file and must be one of the following:

- Indicates an ordinary file.

b Indicates a block special device such as a hard or floppy disk. Hard and floppy disks can be treated as both block and character special devices.

c Indicates a character special device such as a lineprinter or terminal.

d Indicates a directory.

m Indicates a shared data file.

n Indicates a name special file.

p Indicates a named pipe.

s Indicates a semaphore.

From left to right, the next nine characters are interpreted as three sets of three permissions. Each set of three indicates the following permissions:

- Owner permissions,
- Group permissions, and
- All other user permissions.

Within each set, the three characters indicate permission to read, to write, and to execute the file as a command, respectively. For a directory, "execute" permission means permission to search the directory for any files or directories.

Ordinary file permissions have the following meanings:

r	The file is readable.
w	The file is writable.
x	The file is executable.
-	The permission is not granted.

For directories, permissions have the following meanings:

r	Files can be listed in the directory; the directory must also have "x" permission.
w	Files can be created or deleted in the directory. As with "r", the directory itself must also have "x" permission.
x	The directory can be searched. A directory must have "x" permission before you can move to it with the **cd** command, access a file within it, or list the files in it. Remember that a user must have "x" permission to do anything useful to the directory.

The following are some typical directory permission combinations:

d--------	No access at all. This is the mode that denies access to the directory to all users but the super user.
drwx------	Limits access to the owner. The owner can list the contents of this directory and the files in it (if they have appropriate permissions), **cd** to the directory, and add files to, and delete files from, the directory. This is the typical permission for the owner of a directory.
drwxr-x---	In addition to allowing the owner all of the above access permissions, this setting allows group members to list the contents of this directory and files within it and to **cd** to this directory. However, group members cannot create files in, or delete files from, this directory. This is the typical permission an owner gives to others who need access to files in his or her directory.
drwxr-x--x	In addition to allowing the owner and the group all of the above access permissions, this setting allows users other than the owner or members of the group to **cd** to this directory. However, because the **r** is not set for others, other users cannot list the contents of this directory with any of the **ls** commands. This mode is rarely used, but it can be useful if you want to give someone access to a specific file in a directory without revealing the presence of other files in the directory.

The */etc* directory contains files whose permissions vary. Examine the permissions of the files in this directory by entering the following command:

```
l /etc | more
```

Changing File Permissions

The **chmod** command changes the read, write, execute, and search permissions of a file or directory. It has the form:

chmod *instruction filename*

The *instruction* argument indicates which permissions you want to change for which class of users. There are three classes of users, and three levels of permissions. The users are specified as follows:

u User, the owner of the file or directory.

g Group, the group the owner of the file belongs to.

o Other, all users of the system who are not in u or g.

a All users of the system.

The permissions are specified as follows:

r Read, which allows permitted users to look at but not change or delete the file.

w Write, which allows permitted users to change or even delete the file.

x Execute, which allows permitted users to execute the file as a command.

Use **cd** to move to your home directory. Then enter the following command to create *tempfile*:

touch tempfile

The permissions on *tempfile* are probably:

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

Verify this by entering the following command:

l tempfile

Enter the following command to give yourself (the file's owner) execute permissions on *tempfile*:

chmod u+x tempfile

Verify the permissions change with the **l** command. (Of course, since *tempfile* is neither a binary nor a script, having execute permission on it is meaningless.)

Enter the following command to give the group and other users write permission on *tempfile*:

chmod go+w tempfile

Verify the permissions change with the **l** command.

The **chmod** command also allows you to remove permissions. For example, enter the following command to prohibit others from writing to *tempfile*:

chmod o-w tempfile

Remove *tempfile* with the following command:

rm tempfile

Changing Directory Permissions

Directories also have an execute permission, even though they cannot be executed in the same way that a script or binary file can. For directories, the execute attribute is needed in order to do any useful work in a directory. Users who do not have execute permission for a directory cannot **cd** to the directory, list the names of files in the directory, or copy files to or from the directory.

The permissions on your home directory are probably set to the following:

drwxr-xr-x

Verify this by entering the following command:

l -d $HOME

You probably see output like the following:

```
drwxr-xr-x   4 markt    pub          240 Feb 10 09:09 /u/markt
```

This setting allows you, the directory's owner, to **cd** to the directory, to list the contents of the directory and of the files within it (if the file permissions also allow), and to create and delete files in the directory. This setting also allows members of the group and other users to **cd** to the directory, to list the directory's contents and also the contents of files within the directory, if file permissions allow.

To deny any useful access to others, enter the following command:

chmod o-x $HOME

Verify that the permissions were changed with the following command:

l -d $HOME

Your output should look like the following:

```
drwxr-xr--   4 markt    pub          240 Feb 10 09:09 /u/markt
```

Now, only you and members of the group have access to your directory. If you want to restore access to your home directory to other users, enter the following command:

chmod o+x $HOME

Summary

Directories and *subdirectories* are created to organize the XENIX filesystem. Each directory or subdirectory can contain both files and other directories, and can be accessed by anyone with read permission for the file or directory in question.

Whenever you move to a different directory, it becomes, by definition, your *working directory*. There are XENIX commands for:

- Displaying the name of the working directory,
- Creating directories,
- Removing directories,
- Renaming directories,
- Copying directories,
- Listing directory contents,
- Changing your working directory.

The *files* that reside in directories are the most basic means of storing data on a XENIX system. Each file "belongs" to a directory somewhere on the system; it is impossible for a file to exist without a "parent" directory.

You can manipulate files in the same ways that you can manipulate directories (described above). There are XENIX commands for:

- Creating files,
- Deleting files,
- Displaying files,
- Combining files,
- Renaming files,
- Moving files,
- Copying files,
- Printing files.

You can also set the access permissions for any of your files or directories. This feature gives you control over who can read, edit, and execute your files and directories.

Summary

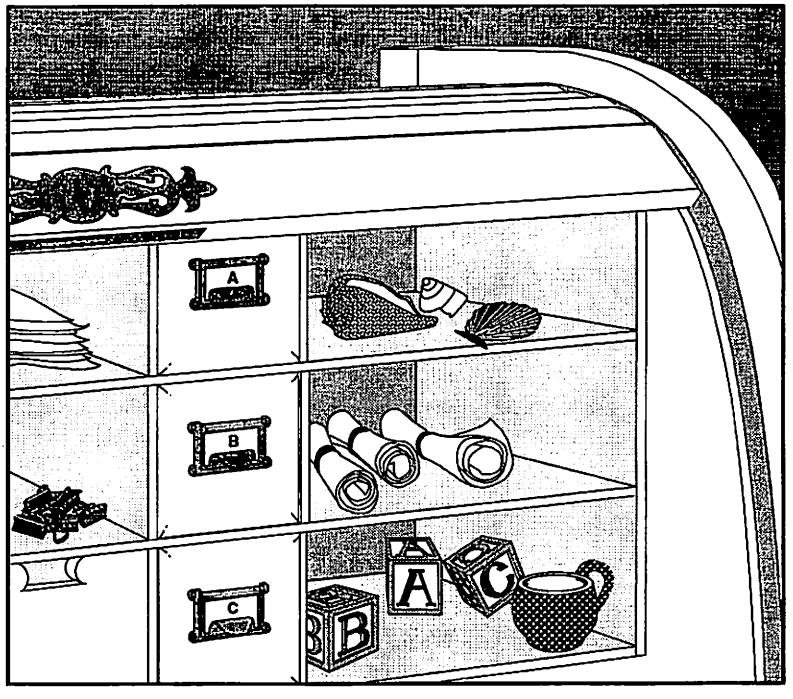
XENIX systems provide a full-screen text editor that is very useful for editing files. This editor, called **vi**, can be called up from anywhere on the system. While you are in **vi**, you can add or delete text to or from a file, change the existing text, or create a new file.

If you have access to a printer, you can produce a hard copy of any of your files. If you have more than one printer, you can choose which one to use. You can also specify the number of copies to be printed, check the status of a print request, and cancel a print request at any time.

XENIX systems provide a set of *utilities* that let you process the information in text files. These utilities enable you to:

- Count the characters in a file,
- Count the words in a file,
- Count the lines in a file,
- Compare the contents of two files,
- Sort files
- Search for patterns in a file.

For a complete description of the commands and utilities presented in this chapter, see the *XENIX User's Guide* and the *XENIX Reference.*



Chapter 5

Housekeeping

Introduction

This chapter explains how to perform "housekeeping" tasks on a XENIX system. Housekeeping tasks are maintenance tasks that you perform periodically, tasks that provide you with information about system resources, as well as tasks that let you operate more efficiently in the XENIX environment. This chapter explains how to perform the following housekeeping tasks:

- Create backups of valuable files and directories,
- Extract files from backup media,
- Make copies of floppy diskettes,
- Find out who is on the system,
- Determine how much disk space is used/free,
- Run a command in the background,
- Delay and kill the execution of commands,
- Use the shell programming language to automate tedious tasks.

This chapter is designed as a tutorial. The best way to use this chapter is to read it at your terminal, entering commands as instructed in the examples.

None of the commands described in this chapter is described in great detail. For a complete explanation of each command, refer to the *XENIX Reference*.

Making Backups

Backing up all the filesystems on a XENIX system is often the responsibility of the system administrator. However, individual users often find it useful to create periodic backups of the particular files with which they work. These backups are created with the **tar** command.

Floppy diskettes are the backup media used most often. Cartridge tapes are also used for backups. However, floppy diskettes and some cartridge tapes must be formatted before they can be used. The sections that follow explain how to format floppy diskettes and certain cartridge tapes and how to use **tar** to create backups.

Formatting Diskettes and Tapes

To format a 5.25 inch 1.2 megabyte diskette (double-sided, double-density) in the first floppy drive, enter the following command:

format

You are instructed to insert a diskette into the drive and press ⟨Return⟩.

If your first floppy drive is a 5.25 inch double-sided high-density drive, enter the following command to format a 1.2 megabyte diskette in it:

format /dev/rfd096ds15

To format a 3.5 inch 720K micro-floppy diskette in the first floppy drive, enter:

format /dev/rfd096ds9

By replacing the **0** that follows the **rfd** in these commands with a **1**, you can format diskettes in a second floppy drive on your system.

It is not necessary to format most cartridge tapes. However, cartridge tapes used with the *mini tape drives* must be formatted. To format one of these cartridges, enter:

format /dev/rctmini

Using tar to Create Backups

The **tar** command is used to create backups. The syntax of **tar** is:

tar [*key*] [*files*]

The *key* argument controls the actions of **tar**. The *files* argument specifies which files to back up.

The *keys* used most often are:

- **c** Creates a backup.
- **x** Extracts files from backup media.
- **t** Lists the contents of backup media.
- **v** Displays the name of each file being processed.
- **f** Creates backups on a specified device.
- **u** Creates a backup only if the file has not been backed up before, or if the file has been modified since the last backup.

Creating a Backup

The steps outlined below explain how to back up all of the files in your home directory onto floppy diskettes. Experiment with **tar** by following these steps.

To back up a different directory, merely **cd** to that directory and follow these same steps. To back up onto a tape, substitute the special device file associated with the tape, such as */dev/rctmini* or */dev/rct0*, for the floppy device file listed in these commands.

1. Log in on the console. This allows you to work within arm's reach of the floppy drive.

2. Determine how many floppy diskettes you need and format that many, using the **format** command as described in "Formatting Diskettes and Tapes" above. To determine how many diskettes to format, enter the following command:

   du -a

   Your output should look like the following:

```
12    ./1.intro.s
74    ./2.concepts.s
14    ./2.concepts.err
0     ./.err
60    ./5.house.s
32    ./3.log.s
2     ./err
2     ./0.title
30    ./6.desk.s
112   ./4.files.s
12    ./4.files.err
4     ./3.log.err
356   .
```


   The number at the bottom represents the total number of 512 byte blocks used by the files in the current directory. In this example, you need a total of 512 x 356 bytes, or roughly 183 kilobytes. You only need to format a single floppy disk to backup this directory.

3. Enter the following command to back up all of the files in your home directory to 5.25 inch 1.2 megabyte floppy diskettes in the first floppy drive:

   tar cvf /dev/fd096ds15 .

   If **tar** requires more than 1 diskette, you are prompted to insert another "volume." Insert another diskette if you see this prompt. The **tar** command is finished when the shell prompt reappears.

To make a backup of just a single file onto a 1.2 megabyte diskette, enter:

tar cvf /dev/fd096ds15 *./filename*

Note that *filename* is preceded by a dot and a slash (**./**). This tells **tar** to treat *filename* as a "relative" rather than an "absolute" filename. (For more information, see **tar**(C).)

tar will recreate, on the backup media, all subdirectories of the directory you are backing up. Thus, if you have a */bin* directory in your home directory, **tar** will create a backup of it, and all the files in it, on the backup media.

Listing the Contents of Backups

To list the contents of a 5.25 inch 360 kilobyte tar-floppy in the first floppy drive, enter:

tar tvf /dev/fd048ds9

To list the contents of a 5.25 inch 1.2 megabyte tar-floppy in the first floppy drive, enter:

tar tvf /dev/fd096ds15

To list the contents of a 3.5 inch tar micro-floppy in the first floppy drive, enter:

tar tvf /dev/fd0135ds9

Experiment with this **tar** option by placing the backup of your home directory that you created in the previous section into the first floppy drive. Enter the appropriate command to list its contents.

Extracting Files from Backups

We recommend that you extract files from backup media into a temporary directory on the hard disk. Once in the temporary directory, you can use the **mv** command to move the extracted files to the correct location on the filesystem. The reason for using a temporary directory is that **tar** will overwrite any files on the hard disk that have the same name as the file being extracted from the backup media. This can result in files being overwritten accidentally.

To extract all of the tar-format files from a 5.25 inch 360 kilobyte diskette in the first floppy drive, enter:

tar xvf /dev/fd048ds9

To extract tar-format files from a 5.25 inch 1.2 megabyte floppy diskette in the first floppy drive, enter:

tar xvf /dev/fd096ds15

To extract a single file from a 1.2 megabyte floppy in the first floppy drive, enter:

tar xvf /dev/fd096ds15 *./filename*

Note that *filename* is preceded by a dot followed by a slash (**./**). This assumes that *filename* was copied to the **tar** floppy diskette with a dot (**.**), as in the examples in "Using tar to Create Backups." When you copy files to a **tar** floppy with a dot, the **./** is prefixed to the filenames. Because you must enter a filename exactly as it appears on the floppy when extracting a file with **tar**, you must enter **./filename** if *filename* was copied to the **tar** floppy with a dot.

Experiment with these **tar** commands by placing the diskette containing the backup of your home directory (that you created in "Using tar to Create Backups") into the first floppy drive. Follow these steps:

1. Enter the following command to change to */tmp*:

   cd /tmp

2. Create a subdirectory in */tmp* by entering:

   mkdir *login*

   Replace *login* with your login name.

3. Enter:

   cd *login*

4. If you are a Bourne or Korn shell user, and if your first floppy drive is a 1.2 megabyte drive, experiment with extracting a single file by entering the following command to extract *.profile*:

   tar xvf /dev/fd096ds15 ./.profile

   If you are a C-shell user, enter:

   tar xvf /dev/fd096ds15 ./.login

   If your floppy drive is not a 1.2 megabyte drive, enter the appropriate special device filename.

5. To check that the files were actually copied to the hard disk, enter:

   lc -a

   The **-a** option tells **lc** to show hidden files, those whose filenames begin with a dot (**.**).

6. To experiment with extracting all of the files on a **tar** floppy, enter the following command if your first floppy drive is a 1.2 megabyte drive:

   tar xvf /dev/fd096ds15

   If your floppy drive is not a 1.2 megabyte drive, enter the appropriate special device filename.

Shorthand tar Notation

The **tar** command also provides shorthand notation. This notation allows you to specify numbers in place of full special device filenames. The file */etc/default/tar* assigns numbers to the various floppy and tape devices. Enter the following to display the contents of */etc/default/tar*:

more /etc/default/tar

Your output should look similar to the following:

```
#   device                  block   size   tape
archive0=/dev/rfd048ds9     18      360    n
archive1=/dev/rfd148ds9     18      360    n
archive2=/dev/rfd096ds15    10      1200   n
archive3=/dev/rfd196ds15    10      1200   n
archive4=/dev/rfd096ds9     18      720    n
archive5=/dev/rfd196ds9     18      720    n
archive6=/dev/rfd0135ds18   18      1440   n
archive7=/dev/rfd1135ds18   18      1440   n
archive8=/dev/rct0          20      0      y
archive9=/dev/rctmini       20      0      y
#   The default device...
archive=/dev/rfd096ds15     10      1200   n
```

This file assigns **0** to the first 360 kilobyte drive, **1** to the second 360 kilobyte drive, **2** to the first 1.2 megabyte drive, **3** to the second 1.2 megabyte drive, and so forth.

To copy all the files in the current directory to 5.25 inch 360 kilobyte diskettes in the first floppy drive, enter:

tar cv .

(The default device is device **0**. You do not have to specify the default device name in the command in order to use the default device.)

To copy all the files in the current directory to 5.25 inch 1.2 megabyte diskettes in the first floppy drive, enter:

tar cv2 .

To extract *filename* from a 3.5 inch 720 kilobyte tar micro-floppy in the first floppy drive, enter:

tar xv4 *./filename*

Note that the version of */etc/default/tar* on your system may differ from that shown here. This is because your system administrator may have edited it. Before you use this shorthand notation, double-check the assignments in the */etc/default/tar* file on your system.

Copying Diskettes

To protect against the loss of data stored on floppy disks, you can use the **diskcp**(C) command to make copies of your floppy diskettes.

You must copy information onto formatted disks. If you format floppies on a XENIX system, you can use them over again without reformatting. If you have disks that have been formatted under another operating system, you must reformat them on a XENIX system before you can use them to make copies of XENIX disks. Be aware that floppies formatted under some operating systems cannot be used under other operating systems, even with reformatting.

The **diskcp** command can format floppies before making copies. The following steps explain how to use **diskcp:**

1. Place the disk that you want to copy, the "source" floppy, in your primary floppy drive. If you created a backup of your home directory, as instructed in "Using tar to Create Backups," experiment with **diskcp** by using this backup. Place it in the floppy drive.

2. Place another floppy in the other drive. This floppy is the "target" disk. Note that any information already on the target disk will be destroyed.

   If you have only one disk drive, leave the source floppy in the drive. It will be copied to the computer's hard disk, and from there to the target floppy. You will be prompted to remove the source floppy and insert the target floppy.

3. To format the target floppy as a 1.2 megabyte floppy before copying, enter the command:

   diskcp -f

   If you do not need to format the target floppy, and if the floppy you are copying is a 1.2 megabyte floppy, enter:

   diskcp

If your computer has dual floppy drives, enter the following command to copy a 1.2 megabyte source floppy directly on a formatted target floppy:

diskcp -d

4. Follow the instructions as they appear on your screen. Note that, with a single drive system, you are prompted to remove the source disk and insert the target disk.

5. If you made a copy of the backup of your home directory, place this floppy in the first floppy drive and verify that the files were copied correctly by entering:

   tar tvf /dev/fd096ds15

   If your floppy is a 360 kilobyte floppy, enter:

   tar tvf /dev/fd048ds9

   Note that you can use the shorthand **tar** notation in these commands, as explained in "Shorthand tar Notation" above.

Getting Status Information

Because a XENIX system is a large, self-contained computing environment, there are many things that you may want to find out about the system itself, such as who is logged in and how much disk space is available. The sections that follow explain how to do this.

Finding Out Who Is on the System

The **who** command lists the names, terminal line numbers, and login times of all users currently logged onto the system. Enter the following command to find out who is on your system:

who

Your output should look like the following:

```
arnold  ttyla  Apr7  10:02
daphne  ttylb  Apr7  07:47
elliot  ttylc  Apr7  14:21
ellen   tty2a  Apr7  08:36
gus     tty2b  Apr7  09:55
adrian  tty2c  Apr7  14:21
```

The **finger** command can also be used to find out who is on the system. It provides more detailed information. To use it, simply enter **finger**.

Determining Disk Usage

The **df** command displays figures on disk free space. When used without options, it reports the number of free blocks and inodes in all the filesystems on your computer. A block is 512 bytes, and an inode is a data structure reserved for information about a file. Enter the following command to display disk free space figures:

df

Your output should look like the following:

```
/        (/dev/root ):     5956 blocks     1437 inodes
```

This means that in the */dev/root* filesystem, there are 5956 blocks and 1437 inodes free. 5956 blocks is roughly equivalent to 3 megabytes.

When used with the **-v** option, **df** reports on the percent of blocks used as well as the number of blocks used and free. Enter the following command:

df -v

Your output should look like the following:

```
Mount Dir   Filesystem   blocks   used    free    % used
/           /dev/root     80152   70192    9960      88%
/y          /dev/y        82194   34314   47880      42%
/u          /dev/u        50000   37840   12160      76%
```

This output indicates that on the */dev/root* filesystem 88% of the blocks, or 70192 blocks out of a total of 80152, are used. 9960 blocks are still free.

Controlling Processes

A command that is executing is considered a *process*. On a XENIX system, a user can run several processes at the same time, one in the *foreground* and several others in the *background*. The foreground process is the one that is currently executing on your terminal. This is the only process that can accept input from your keyboard. For instance, when you are editing with **vi**, the **vi** program is running as a foreground process.

Keyboard input cannot be sent to background processes. It is often useful to execute processes that are time consuming or require no keyboard input in the background. Controlling foreground and background processes is the subject of this section.

Placing a Command in the Background

Normally, commands sent from the keyboard are executed in strict sequence. One command must finish executing before the next command can begin. However, if you place a command in the background, you can continue to enter commands in the foreground, even if the background command is not finished executing.

To place a command in the background, put an ampersand (&) at the end of the command line. For example, enter the following command to create, and then count, the characters in a large file. Note that this command line is two lines long. This is made possible by placing the backslash (\) on the command line before pressing ⟨Return⟩. The backslash tells the shell that the command line continues on the next line:

```
cat /etc/termcap /etc/termcap /etc/termcap > largefile; \
wc -c largefile > characters &
```

The output of the **wc** part of the command is redirected to *characters*. If you issued this command without redirecting the output, the output would print on your screen, no matter what else you might be doing. This can be very disruptive. Redirecting the output of a background command to a file is a simple way of avoiding such disruptions.

Use **cat** to display the contents of *characters*. When you are finished, use **rm** to remove both *largefile* and *characters*.

When commands are placed in the background, you cannot abort them by pressing the **INTERRUPT** key, as you can with foreground commands. You must use the **kill** command to abort a background process. This command is described in "Killing a Process," below.

Delaying the Execution of a Command

In addition to putting commands in the background, you can delay command execution. This is done with the **at** command. This command allows you to set up a series of commands to be executed at a specified time in the future. Use of this command is controlled by the system administrator; you can use it only after he or she has enabled you to do so.

The **at** command accepts standard input. The simplest form of the command is:

at *time day* < *file*

Note

Use of the **at** command is subject to the discretion of the system administrator. See the section "Permitting Users Access to Job Scheduling" in the "Using the Job Scheduling Commands: at, cron and batch" appendix of the *XENIX System Administrator's Guide* for details.

The *file* argument is the name of the file that contains the command or commands to be executed. The *time* argument is the time of day, in digits, followed by "am" or "pm." One- and two-digit numbers are interpreted as hours, three- and four-digit numbers as hours and minutes. You cannot use *time* arguments of more than four digits. The *day* argument is optional. It is either a month name followed by a day number, or a day of the week. If no *day* is specified, the command is executed the next time the specified *time* occurs.

For example, suppose that you have a large file that you want to print, but you don't want to do this during work hours because it will monopolize the printer for a long time. You could use **at** to print the file late at night, when nobody is in the office. To do so, first use **vi** to create a file containing the print command. Call the file *printfile*. This file might contain the following line:

lp *filename*

The *filename* argument is the name of the large file that you want to print.

After you create *printfile*, enter the following command:

```
at 11pm wed < printfile
```

There is no need to place this command in the background. Once you enter it and press ⟨Return⟩, the XENIX prompt reappears. This causes the command in *printfile* to be executed at 11:00 p.m. on Wednesday.

Note that **at** is unaffected by logging out. To display a list of files that you have waiting to be processed with **at**, use the **at -l** command. This command also lists the following information:

- The file's ID number.
- The command invoking the file (**at** or **batch**).
- The date and time the file is processed.

To cancel an **at** command, first check the list of files you have waiting to be processed and note the file ID number. Then use the **at -r** command to remove the file or files from the list.

The **at -r** command has the form:

at -r *ID-number*

For example, the following command removes file number 504510300.a, canceling whatever commands were included in that file:

```
at -r 504510300.a
```

Note that a user can only cancel his or her own files.

The files */usr/lib/cron/at.allow* and */usr/lib/cron/at.deny* control who has access to the **at** command. On many systems, only the super user is allowed to use **at**. Contact your system administrator if you need to use the **at** command but are denied access to it.

Finding Out Which Processes are Running

The **ps** command stands for "processes status" and displays information about currently running processes. This information is crucial if you need to kill a background process.

To display information about commands that you currently have running, enter the following:

ps

Your output should look like the following:

```
  PID   TTY   TIME   COMMAND
   49   2a    0:28   sh
11267   2a    0:00   ps
```

The PID column gives a unique process identification number that can be used to kill a particular process. The TTY column shows the terminal with which the process is associated. The TIME column shows the cumulative execution time for the process. The COMMAND column shows the actual command that is executing.

Enter the following command to display information about all the processes running on the system:

ps -e

To find out about the processes running on a terminal other than the one you are using, use the **-t** option and specify the terminal number. For example, to find out what processes are associated with terminal 2c, enter:

ps -t2c

Killing a Process

To stop execution of a foreground process, press your terminal's **INTERRUPT** key. This is often the ⟨Del⟩ key. Pressing this key will kill whatever foreground process is currently running. To kill all your processes executing in the background, enter the following command:

kill 0

To kill only a specified process executing in the background, you need to enter the following command:

kill *signal_number process_ID_number*

The *signal_number* is optional. It is sometimes needed to kill "stubborn" processes. Even stubborn processes can usually be killed with *signal_number* 9. Find out the *process_ID_number* with the **ps** command.

As an example, try killing the process associated with your shell. Note that when you log in to the system, you are placed in a shell. If you kill this shell process, you log yourself out. Enter **ps** and look at the process ID associated with either **sh** or **csh** in the COMMAND column. Suppose that this number is 4831. To kill your shell, enter the following command:

kill -9 4831

After entering this command, you see the login prompt again. Try it!

To guard against other users having control over your account, you can only kill your own processes.

Note

Killing a process associated with the **vi** editor can result in unpredictable terminal behavior. Also, temporary files that are normally created when a command starts, and then deleted when the command finishes, can be left behind after a **kill** command. Temporary files are normally kept in the directory */tmp*. You should check this directory periodically and delete your old files.

Shell Programming

The Bourne, Korn, and C-shell offer powerful programming features. If you have ever done batch programming in MS-DOS, you have some idea of what shell programming on a XENIX system is like. This section discusses the rudiments of shell programming. For a more complete discussion, see "The Shell," "Korn Shell," and "The C-Shell," in the *XENIX User's Guide*.

In "Placing a Command in the Background," you were instructed to enter the following command:

```
cat /etc/termcap /etc/termcap /etc/termcap > largefile; \
wc -c largefile > characters &
```

However, you could have placed these commands in a file and executed the file.

Try it by creating a file called *command.file* with **vi**. Place the following lines in *command.file*:

```
cat /etc/termcap > largefile
cat /etc/termcap >> largefile
cat /etc/termcap >> largefile
wc -c largefile > characters
```

After placing these four lines in *command.file*, type **:x** to save it and quit **vi**. Now you must enter the following command to make *command.file* executable:

```
chmod +x command.file
```

Finally, execute *command.file* in the background by entering the following command:

```
command.file &
```

After a few moments, enter the following command to verify that *command.file* executed correctly:

```
cat characters
```

In fact, using some of the more sophisticated shell programming features that let you control the flow of a program, you could have written *command.file* as follows:

```
for name in /etc/termcap
do
      cat $name $name $name > largefile
done
wc -c largefile > characters
```

Whenever you place XENIX commands in a file, always remember to use the **chmod** command to make the file executable. (The **chmod** command is discussed in "Using File and Directory Permissions" in Chapter 4 of this tutorial.)

Summary

With XENIX systems, files are automatically stored in the computer's memory. There are provisions for creating backup copies of files, which can be stored on cartridge tapes or floppy diskettes. Once a backup copy has been made, you can display its contents, extract files from it, and run off additional copies. The operating system accepts both 3.5 inch and 5.25 inch diskettes that have any of the commonly-produced memory capacities (360K, 720K, 1.2M, 1.4M).

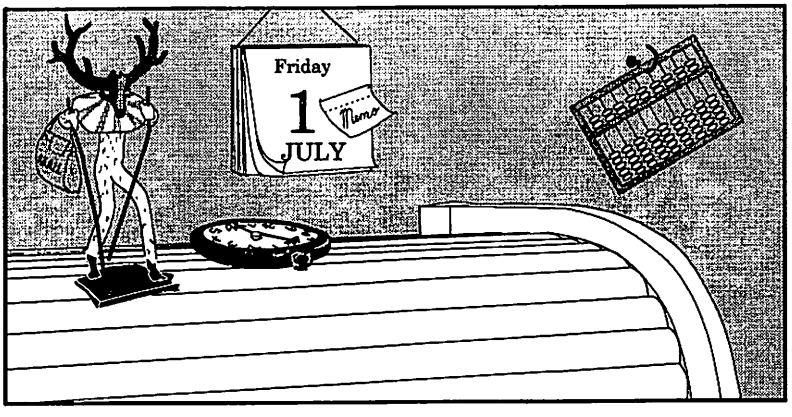
A currently-running command is called a *process*. On XENIX systems, you can run several processes at the same time by putting one in the *foreground* and the rest in the *background*. The foreground process is the one that shows up on your terminal screen, and is the only one that will accept input from your keyboard. When you enter commands, you must specify which ones should be put into the background. If you do not, the operating system assumes that they should all be in the foreground, and will execute them one at a time.

There are several commands for controlling the execution of processes. The commands covered in this chapter allow you to delay the execution of a process until a specified time, and to kill a foreground or background process.

There are also commands that provide information about the operating system itself. The commands presented in this chapter allow you to find out who is logged into the system, how much space is left on the system's main storage disk, and what processes are currently running.

Both the C-shell, Bourne shell, and Korn shell provide environments for writing very useful programs. You can write a *shell program* by creating a text file that contains a series of XENIX commands and then designating the file as executable. After that, the file (program) is executed whenever you enter the filename at the shell prompt.

For details about the commands presented in this chapter, see the *XENIX User's Guide* and the *XENIX Reference*.



Chapter 6

XENIX Desktop Utilities

Introduction

XENIX systems include a series of "desktop" utilities, programs that help you organize your work environment, and programs that allow you to communicate with other users on the system. This chapter describes these utilities, explaining how you can:

- Display the date, time and a calendar,
- Communicate with other users on the system,
- Use the system's automatic reminder service,
- Use the system's interactive calculator.

This chapter is designed as a tutorial. The best way to use this chapter is to read it at your terminal, entering commands as instructed in the examples.

None of the commands described in this chapter is described in great detail. For a complete explanation of each command, refer to the *XENIX Reference*.

Using the System Clock and Calendar

There are commands that display the date and time, as well as commands that display a calendar for virtually any month or year that you choose. The following sections explain these commands.

Finding Out the Date and Time

The **date** command displays the date and time. Enter:

date

Your output should look like the following:

```
Mon Jan 25 08:26:13 PST 1988
```

Displaying a Calendar

The **cal** command displays the calendar of any month or year that you specify. For example, to display the calendar for March 1952, enter:

cal mar 1952

The result is:

```
     March 1952

  S  M Tu  W Th  F  S
                   1
  2  3  4  5  6  7  8
  9 10 11 12 13 14 15
 16 17 18 19 20 21 22
 23 24 25 26 27 28 29
 30 31
```

The most common month abbreviations are accepted. The month can also be expressed as a digit. To display the calendar for an entire year, leave out the month. The year must always be expressed in full. The command **cal 88** displays the calendar for the year 88, not 1988.

Using the Mail Service

Several programs allow you to communicate with other users on a XENIX system. Two of the most useful are **mail** and **write**. The **mail** program allows you to send a message to a user's system "mailbox." The **write** program allows you to write a message directly to a user's terminal, if the user is logged into the system. Both of these programs are described below.

Sending Mail

The **mail** program is a system-wide facility that permits you to exchange mail with other users. Experiment with **mail** by sending a message to yourself. To do so, enter the following command:

mail *login*

Be sure to replace *login* with your login name.

Depending on how your system administrator has configured your mail system, you may see the following prompt:

```
Subject:
```

If you see this prompt, enter a short description of the message to follow. In this case, enter **test**.

You can now enter your message. When you are finished, press ⟨Ctrl⟩d to terminate message entry and mail your message. However, an alternative method is available for composing a message. If you enter ~v, (a tilde, followed by a v) you are placed in **vi**. Once in **vi**, you can compose your message, using all the features available in **vi**. This method of composing a message is much more flexible than the other, as it allows you to correct mistakes in your message before you send it. Correcting mistakes in a message produced only with **mail** often results in control characters cluttering up the message.

If you use **vi** to compose your message, enter **:x** when you are finished, then ⟨Ctrl⟩d to terminate message entry. Again, depending on how your mail system is configured, you may or may not see the following prompt:

```
Cc:
```

If you do see this prompt, enter the names of those users who should receive "carbon copies" of this message. It is often convenient to **Cc:** yourself. Since this is a test message to yourself, just press ⟨Return⟩. You are now returned to the XENIX prompt.

Often you will want to send a text file to other users on the system. You can use the XENIX redirection facility to do this. Suppose that the file you want to send is named *schedule*, that it is in the current directory, and that you want to send it to users Naomi and Bea. To do so, enter the following command at the XENIX prompt:

mail naomi bea < schedule

Sending files this way is fast because you do not have to enter **mail**.

The **mail** facility has many options. These options allow you to include already composed messages in a mailing, to enclose a message you are responding to in your reply, to create mail aliases to send messages to several users at once, and so forth. These options are discussed in detail in "mail" in the *XENIX User's Guide*.

Receiving Mail

When you log in, you may see the following the message:

```
you have mail
```

To read your mail, enter:

mail

A list of message headers is displayed, each with a number in front of it. The list should look similar to the following:

```
1  john  Wed  Sep  21  09:21  26/782 "Notice"
2  sam   Tue  Sep  20  22:55  6/83   "Meeting"
3  tom   Mon  Sep  19  01:23  6/84   "Invite"
```

Reading from left to right, the headers tell you who the message is from, the date and time it was sent, the number of lines and characters in the message, and the message's subject.

To read a message, simply enter the number of the message you want to read, then press ⟨Return⟩. For example, to read the message from sam, enter **2** and then press ⟨Return⟩. Read the message that you sent to yourself, as described in the previous section.

There are several things you can do with your **mail** messages after you read them. You can delete them, save them, and/or respond to them.

To delete a message, press **d**, the message number and then ⟨Return⟩. To save a message in your mailbox, press **ho**, the message number and then ⟨Return⟩. To save a message in *filename* in the current directory, press **s**, the message number, type *filename* and press ⟨Return⟩. If *filename* does not exist, **mail** creates it.

Two commands are available for responding to mail. These are the **r** and **R** commands. If you press **r**, followed by the number of a message, followed by ⟨Return⟩, followed by a response, your response is sent to the author of the message. If you press **R**, followed by the number of a message, followed by ⟨Return⟩, followed by a response, your response is sent to the author of the message plus all users who were on the **Cc:** list of the original message.

After reading a message, you might want to list your message headers again. Do so by entering **h** followed by ⟨Return⟩. If you have more messages than will display on the screen, enter **h+** followed by ⟨Return⟩. This will cause the next 18 message headers to display. To display the previous 18 message headers, enter **h-**.

Note that you can send mail from within the **mail** program. To send mail to a user named "joe" from within **mail**, simply enter the following command:

mail joe

Then follow the steps outlined above in "Sending Mail" to send a message to joe.

To quit **mail**, enter **q** followed by ⟨Return⟩.

Respond to the message that you sent yourself by doing the following:

1. Enter mail by typing **mail**.
2. Enter **r***number*, where *number* is the number of the message that you sent to yourself.
3. Press ⟨Return⟩.
4. Compose your response. Remember that you can enter **~v** to compose your response in **vi**.
5. Press ⟨Ctrl⟩d to terminate your response. If you compose your response in **vi**, you will have to press **:x** to leave **vi** and then ⟨Ctrl⟩d.
6. Press ⟨Return⟩ to send the response.
7. Type **restart**. This will cause **mail** to display any messages that were sent to you while you were in **mail**.
8. You should see your response to the message you sent yourself. Press the number that corresponds to this response to view it.
9. When you are finished looking at your messages, press **q** to leave **mail**.

Writing to a Terminal

The **write** program allows you to send messages directly to another user's terminal. You can reach another user as long as he or she is logged into the system and has not turned off write access to his or her terminal. For example, to write to joe's terminal, enter:

write joe

After you execute this command by pressing ⟨Return⟩, joe sees a message like the following on his terminal:

```
Message from login tty012...
```

The *login* in this message is your login name. To respond, joe enters:

write *login*

Again, *login* is replaced by your login name.

From this point on, each line that you enter is displayed both on your own terminal screen and on joe's. Each line that joe enters is displayed on both his screen and yours. To terminate the writing of text to joe, enter ⟨Ctrl⟩d alone on a line. Joe has to do the same to terminate his **write** session with you.

A typical procedure for coordinating communication in a two-way **write** is for each party to end each line with a distinctive signal, normally (o) for "over." The last line of a message is often followed by (oo) for "over and out."

Experiment with **write** by sending a message to yourself. Do so by entering the following command:

write *login*

Replace *login* with your login name. You should see a message like the following:

```
Message from login ttynn...
```

Now simply enter your message. Since you are writing to yourself, everything you enter appears on *your* screen twice.

For example, your **write** session might look like the following:

```
Hello Mark o
Hello Mark o
Remember, we have a meeting at 12:00. o
Remember, we have a meeting at 12:00. o
Right, see you there. oo
Right, see you there. oo
```

Press ⟨Ctrl⟩d to terminate the **write** session.

If you do not want to be interrupted by a message, enter the command **mesg n** any time after you have logged in. Anyone that tries to write to you after you have entered this command is denied permission. If you want to once again be accessible via the **write** program, enter **mesg y**. If you want write access to automatically be turned off whenever you log in, put the **mesg n** command in your *.login* file.

Using the Automatic Reminder Service

An automatic reminder service is available to all users. Once each day, the operating system automatically searches each user's home directory for a file named *calendar*, the contents of which might look like the following:

```
1/23 David's wedding
2/9  Mira's birthday
3/30 Paul's birthday
4/27 Meeting at 2:00
9/1  Karen's birthday
10/3 License renewal
```

Each line of *calendar* is examined. Lines containing today's and tomorrow's dates are extracted and mailed to you. To look at these reminders, you must invoke **mail**.

The file *calendar* is not created for you automatically. You have to create it yourself if you want to use this reminder service. It must be in your home directory.

Use **vi** or any other XENIX text editor to create and edit *calendar*. Be sure to place each date/event entry on a separate line. Dates can be specified in a variety of formats. Any of the following is acceptable:

9/7
Sep. 7
september 7

Using the Calculator

The **bc** command invokes an interactive desktop calculator that can be used like a hand-held calculator. A typical session with **bc** is shown below. Note that the session is begun by entering **bc** at the XENIX prompt and ended by entering **quit** on a line by itself. Comments explain what action is performed after each input line.

Action	Comment
bc	Activate bc
123.456789 + 987.654321	Add and output
1111.111110	
9.0000000 - 9.0000001	Subtract and output
-.0000001	
64/8	Divide and output
8	
1.12345678934 * 2.3	Note precision
2.58395061548	
19%4	Find remainder
3	
3^4	Exponentiation
81	
2/1*2	Note precedence
4	
2/(1*2)	Note precedence again
1	
x = 46.5	Assign value to x
y = 52.5	Assign value to y
x + y + 1.0000	Add and output
100.0000	
obase=16	Set hex output base
15	Convert to hex
F	
16	Convert to hex
10	
64	Convert to hex
40	
255	Convert to hex
FF	
256	Convert to hex
100	
512	Convert to hex
200	
quit	Must type whole word

Also available are scaling, function definition, and programming statements much like those in the C programming language. Other features include assignment to named registers and subroutine calling. For more information, see the **bc**(C) manual page.

Summary

XENIX systems have several utilities that help you organize your work environment and communicate with other users on the system. There are XENIX utilities for:

- Displaying the current date and time,
- Displaying a calendar for any month of any year,
- Reminding you of upcoming events,
- Invoking an interactive calculator.

XENIX systems include a electronic *mail* system that lets you send and receive messages to and from other users. When you send mail to someone, your message is stored in the recipient's *mailbox* until he or she retrieves it. By using this format, the mail utility does not interrupt other users, and allows messages to be sent to users who are not currently logged in.

In addition to the **mail** utility, there is a *write* utility that lets you communicate directly with another user. To use this utility, both the sender and the recipient must be logged in. As each person types a message, it immediately appears on both users' terminals.

For details about the commands and utilities presented in this chapter, see the *XENIX User's Guide* and the *XENIX Reference*.

Index

Special Characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

SCO® XENIX® System V

Operating System

User's Guide

The Santa Cruz Operation, Inc.

Document Version 2.3.4C
Date: 28 March 1991

Contents

Chapter 1

Introduction

Overview

This guide provides extensive information on several of the most useful XENIX facilities, including **mail**, the **vi** text editor, **uucp**, and **micnet**. In addition, the guide includes information on the four XENIX "shells": the Bourne shell, the Korn Shell, the Visual shell, and the C-shell.

About This Guide

This guide is organized as follows:

Chapter 1, "Introduction" provides an overview of the contents of this guide and gives a list of the notational conventions used throughout.

Chapter 2, "vi: A Text Editor" explains how to use the XENIX fullscreen editor, vi.

Chapter 3, "mail" explains how to use the XENIX electronic mail facility.

Chapter 4, "Communicating with Other Sites" explains how to transfer files to and from and how to execute commands on other computer sites. These other sites might be UNIX or XENIX sites, but they do not need to be. They can, for instance, be MS-DOS™ sites.

Chapter 5, "The Shell" explains how to use the XENIX Bourne shell.

Chapter 6, "The C-Shell" explains how to use the features of the XENIX C shell.

Chapter 7, "The Korn Shell" explains how to use the powerful features of the XENIX Korn shell.

Chapter 8, "Using The Visual Shell" explains how to use the menu-driven Visual shell.

Notational Conventions

This guide uses a number of notational conventions to describe the syntax of XENIX commands:

Initial Capitals	Initial Capitals indicate the name of a command or mode. When a command is introduced it is followed by the keystroke that invokes it, (i.e. the Insert (**i**) command).
boldface	Boldface indicates a command, option, flag, or program name to be entered as shown. Keystrokes are boldfaced when they indicate a command to enter as shown, (i.e. enter the **i** command and press ⟨Return⟩). Commands that are issued while within a program, such as a file editor like **vi**(C), are not boldfaced so they will not be confused with commands given to the shell. Boldface indicates the name of a XENIX utility or library routine. (To find more information on a given utility, consult the "Alphabetized List" in the appropriate *Reference* for the manual page that describes it.)

italics	Italics indicate a filename. This pertains to library include filenames (i.e. *stdio.h*), as well as, other filenames (i.e. */etc/ttys*).
	Italics indicate a placeholder for a command argument. When entering a command, a placeholder must be replaced with an appropriate filename, number, or option.
	Italics indicate a specific identifier, supplied for variables and functions, when mentioned in text.
	Italics indicate a reference to part of an example.
	Italics indicate emphasized words or phrases in text.
`screen font`	This font is used for screen displays and messages.
[ ]	Brackets indicate that the enclosed item is optional. If you do not use the optional item, the program selects a default action to carry out.
	Brackets indicate the position of the cursor in text examples.
...	Ellipses indicate that you can repeat the preceding item any number of times.
	Vertical ellipses indicate that a portion of a program example is omitted.
" "	Quotation marks indicate the first use of a technical term.
	Quotation marks indicate a reference to a word rather than a command.

Chapter 2

vi: A Text Editor

Introduction

Any ASCII text file, such as a program or document, may be created and modified using a text editor. There are two text editors available on XENIX systems, **ed** and **vi**. **ed** is discussed in the "ed" chapter of this manual.

vi (which stands for "visual") combines line-oriented and screen-oriented features into a powerful set of text editing operations that will satisfy any text editing need.

The first part of this chapter is a demonstration that gives you some hands-on experience with **vi**. It introduces the basic concepts you must be familiar with before you can really learn to use **vi**, and shows you how to perform simple editing functions. The second part is a reference that shows you how to perform specific editing tasks. The third part describes how to set up your **vi** environment and how to set optional features. The fourth part is a summary of commands.

Because **vi** is such a powerful editor, it has many more commands than you can learn at one sitting. If you have not used a text editor before, the best approach is to become thoroughly comfortable with the concepts and operations presented in the demonstration section, then refer to the second part for specific tasks you need to perform. All the steps needed to perform a given task are explained in each section, so some information is repeated several times. When you are familiar with the basic **vi** commands you can easily learn how to use the more advanced features.

If you have used a text editor before, you may want to turn directly to the task-oriented part of this chapter. Begin by learning the features you will use most often. If you are an experienced user of **vi** you may prefer to use **vi**(C) in the *XENIX Reference* instead of this chapter.

This chapter covers the basic text editing features of **vi**. For more advanced topics, and features related to editing programs, refer to **vi**(C) in the *XENIX Reference*.

Demonstration

The following demonstration gives you hands-on experience using **vi**, and introduces some basic concepts that you must understand before you can learn more advanced features. You will learn how to enter and exit the editor, insert and delete text, search for patterns and replace them, and how to insert text from other files. This demonstration should take one hour. Remember that the best way to learn **vi** is to actually use it, so don't be afraid to experiment.

Before you start the demonstration, make sure that your terminal has been properly set up. See the section "Setting the Terminal Type," for more information about setting up your terminal for use with **vi**.

Entering the Editor

To enter the editor and create a file named *temp*, enter:

vi temp

Your screen will look like this:

```
~
~
~
~
~
~
~
~
~
"temp" [New file]
```

Note that we show a twelve-line screen to save space. In reality, **vi** uses whatever size screen you have.

You are initially editing a copy of the file. The file itself is not altered until you save it. Saving a file is explained later in the demonstration. The top line of your display is the only line in the file and is marked by the cursor, shown above as an underline character. In this chapter, when the cursor is on a character that character will be enclosed in square brackets ([]).

The line containing the cursor is called the *current line*. The lines containing tildes are not part of the file: they indicate lines on the screen only, not real lines in the file.

Inserting Text

To begin, create some text in the file *temp* by using the Insert (i) command. To do this, press:

```
i
```

Next, enter the following five lines to give yourself some text to experiment with. Press ⟨Return⟩ at the end of each line. If you make a mistake, use the ⟨Bksp⟩ key to erase the error and enter the word again.

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Press the ⟨Esc⟩ key when you are finished.

Like most **vi** commands, the **i** command is not shown (or "echoed") on your screen. The command itself switches you from Command mode to Insert mode.

When you are in Insert mode every character you enter is displayed on the screen. In Command mode the characters you enter are not placed in the file as text; they are interpreted as commands to be executed on the file. If you are not certain which mode you are in, press ⟨Esc⟩ until you hear the bell. When you hear the bell you are in Command mode.

Once in Insert mode, the characters you enter are inserted into the file; they are *not* interpreted as **vi** commands. To exit Insert mode and reenter Command mode you will always press ⟨Esc⟩. This switching between modes occurs often in **vi**, and it is important to get used to it now.

Repeating a Command

Next comes a command that you will use frequently in vi: the Repeat command. The Repeat command repeats the most recent Insert or Delete command. Since we have just executed an Insert command, the Repeat command repeats the insertion, duplicating the inserted text. The Repeat command is executed by entering a period (.) or "dot" . So, to add five more lines of text, enter ".". The Repeat command is repeated relative to the location of the cursor and inserts text *below* the current line. (Remember, the current line is always the line containing the cursor.) After you enter dot (.), your screen will look like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
```

Undoing a Command

Another command which is very useful (and which you will need often in the beginning) is the Undo (**u**) command. Press

u

and notice that the five lines you just finished inserting are deleted or "undone".

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```


Now enter:

u

again, and the five lines are reinserted! This undo feature can be very useful in recovering from inadvertent deletions or insertions.

Moving the Cursor

Now let's learn how to move the cursor around on the screen. In addition to the arrow keys, the following letter keys also control the cursor:

h Left

l Right

k Up

j Down

The letter keys are chosen because of their relative positions on the keyboard. Remember that the cursor movement keys only work in Command mode.

Try moving the cursor using these keys. (First make sure you are in Command mode by pressing the ⟨Esc⟩ key.) Then, enter the **H** command to place the cursor in the upper left corner of the screen. Then enter the **L** command to move to the lowest line on the screen. (Note that case is significant in our example: **L** moves to the lowest line on the screen; while **l** moves the cursor forward one character.) Next, try moving the cursor to the last line in the file with the goto command, **G**. If you enter **2G**, the cursor moves to the beginning of the second line in the file; if you have a 10,000 line file, and enter **8888G**, the cursor goes to the beginning of line 8888. (If you have a 600 line file and enter **800G** the cursor does not move.)

These cursor movement commands should allow you to move around well enough for this demonstration. Other cursor movement commands you might want to try out are:

w	Moves forward a word
b	Backs up a word
0	Moves to the beginning of a line
$	Moves to the end of a line

You can move through many lines quickly with the scrolling commands:

⟨Ctrl⟩u	Scrolls up 1/2 screen
⟨Ctrl⟩d	Scrolls down 1/2 screen
⟨Ctrl⟩f	Scrolls forward one screenful
⟨Ctrl⟩b	Scrolls backward one screenful

Deleting

Now that we know how to insert and create text, and how to move around within the file, we are ready to delete text. Many Delete commands can be combined with cursor movement commands, as explained below. The most common Delete commands are:

dd	Deletes the current line (the line the cursor is on), regardless of the location of the cursor in the line.
dw	Deletes the word above the cursor. If the cursor is in the middle of the word, deletes from the cursor to the end of the word.
x	Deletes the character above the cursor.
d$	Deletes from the cursor to the end of the line.
D	Deletes from the cursor to the end of the line.
d0	Deletes from the cursor to the start of the line.
.	Repeats the last change. (Use this only if your last command was a deletion.)

To learn how all these commands work, we will delete various parts of the demonstration file. To begin, press ⟨Esc⟩ to make sure you are in Command mode, then move to the first line of the file by entering:

```
1G
```

At first, your file should look like this:


```
[F]iles contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
```

To delete the first line, enter:

```
dd
```

Your file should now look like this:

```
[T]ext contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
```

Delete the word the cursor is sitting on by entering:

```
dw
```

After deleting, your file should look like this:

```
[c]ontains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
```


You can quickly delete the character above the cursor by pressing:

```
x
```

This leaves:

```
[o]ntains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
```

Now enter a w command to move your cursor to the beginning of the word *lines* on the first line. Then, to delete to the end of the line, enter:

```
d$
```

Your file looks like this:

```
ontains_
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
```

To delete all the characters on the line *before* the cursor enter:

d0

This leaves a single space on the line:

```
_
Lines contain characters.
Files contain text.
Text contains lines.
Characters form words.
Words form text.
Lines contain characters.
Characters form words.
Words form text.
~
~
```

For review, let's restore the first two lines of the file.

Press **i** to enter Insert mode, then enter:

Files contain text.
Text contains lines.

Press ⟨Esc⟩ to go back to Command mode.

Searching for a Pattern

You can search forward for a pattern of characters by entering a slash (/) followed by the pattern you are searching for, terminated by a ⟨Return⟩. For example, make sure you are in Command mode (press ⟨Esc⟩), then press

H

to move the cursor to the top of the screen. Now, enter:

/char

Do not press ⟨Return⟩ yet. Your screen should look like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
/char_
```

Press ⟨Return⟩. The cursor moves to the beginning of the word *characters* on line three. To search for the next occurrence of the pattern *char*, press **n** (as in "next"). This will take you to the beginning of the word *characters* on the eighth line. If you keep pressing "n" **vi** searches past the end of the file, wraps around to the beginning, and again finds the *char* on line three.

Note that the slash character and the pattern that you are searching for appear at the bottom of the screen. This bottom line is the **vi** status line.

The *status line* appears at the bottom of the screen. It is used to display information, including patterns you are searching for, line-oriented commands (explained later in this demonstration), and error messages.

For example, to get status information about the file, press ⟨Ctrl⟩g. Your screen should look like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain [c]haracters.
Characters form words.
Words form text.
~
"temp" [Modified] line 4 of 10 --4%--
```


The status line on the bottom tells you the name of the file you are editing, whether it has been modified, the current line number, the number of lines in the file, and your location in the file as a percentage of the number of lines in the file. The status line disappears as you continue working.

Searching and Replacing

Let's say you want to change all occurrences of *text* in the demonstration file to *documents*. Rather than search for *text*, then delete it and insert *documents*, you can do it all in one command. The commands you have learned so far have all been *screen-oriented*. Commands that can perform more than one action (searching and replacing) are *line-oriented* commands.

Screen-oriented commands are executed at the location of the cursor. You do not need to tell the computer where to perform the operation; it takes place relative to the cursor. *Line-oriented* commands require you to specify an exact location (called an "address") where the operation is to take place. Screen-oriented commands are easy to enter, and provide immediate feedback; the change is displayed on the screen. Line-oriented commands are more complicated to enter, but they can be executed independent of the cursor, and in more than one place in a file at a time.

All line-oriented commands are preceded by a colon which acts as a prompt on the status line. Line-oriented commands themselves are entered on this line and terminated with a ⟨Return⟩.

In this chapter, all instructions for line-oriented commands will include the colon as part of the command.

To change *text* to *documents*, press ⟨Esc⟩ to make sure you are in Command mode, then enter:

```
:1,$s/text/documents/g
```

This command means "From the first line (1) to the end of the file ($), find *text* and replace it with *documents* (s/text/documents/) everywhere it occurs on each line (g)".

Press ⟨Return⟩. Your screen should look like this:

```
Files contain documents.
Text contains lines.
Lines contain characters.
Characters form words.
Words form documents.
Files contain documents.
Text contains lines.
Lines contain characters.
Characters form words.
[W]ords form documents.
~
~
```

Note that *Text* in lines two and eight was not changed. Case is significant in searches.

Just for practice, use the Undo command to change *documents* back to *text*. Press:

```
u
```

Your screen now looks like this:

```
[F]iles contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
```


Leaving vi

All of the editing you have been doing has affected a copy of the file, and *not* the file named *temp* that you specified when you invoked **vi**. To save the changes you have made, exit the editor and return to the XENIX shell, enter:

```
:x
```

Remember to press ⟨Return⟩. The name of the file, and the number of lines and characters it contains are displayed on the status line:

```
"temp" [New file] 10 lines, 214 characters
```

Then the XENIX prompt appears.

Adding Text From Another File

In this section we will create a new file, and insert text into it from another file. First, create a new file named *practice* by entering:

vi practice

This file is empty. Let's copy the text from *temp* and put it in *practice* with the line-oriented Read command. Press ⟨Esc⟩ to make sure you are in Command mode, then enter:

```
:r temp
```

Your file should look like this:

```
[F]iles contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
```

The text from *temp* has been copied and put in the current file *practice*. There is an empty line at the top of the file. Move the cursor to the empty line and delete it with the **dd** command.

Leaving vi Temporarily

vi allows you to execute commands outside of the file you are editing, such as **date**. To find out the date and time, enter:

```
:!date
```

Press ⟨Return⟩. This displays the date, then prompts you to press ⟨Return⟩ to reenter Command mode. Go ahead and try it. Your screen should look similar to this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
-
:!date
Mon Jan 9 16:33:37 PST 1985
[Press return to continue]_
```


Changing Your Display

Besides the set of editing commands described above, there are a number of options that can be set either when you invoke **vi**, or later when editing. These options allow you to control editing parameters such as line number display, and whether or not case is significant in searches. In this section we will learn how to turn on line numbering, and how to look at the current option settings.

To turn on automatic line numbering, enter:

```
:set number
```

Press ⟨Return⟩. Your screen is redrawn, and line numbers appear to the left of the text. Your screen looks like this:

```
 1  Files contain text.
 2  Text contains lines.
 3  Lines contain characters.
 4  Characters form words.
 5  Words form text.
 6  Files contain text.
 7  Text contains lines.
 8  Lines contain characters.
 9  Characters form words.
10  Words form text.
~
~
```


You can get a complete list of the available options by entering:

```
:set all
```

and pressing ⟨Return⟩. Setting these options is described in the section "Setting Up Your Environment," but it is important that you be aware of their existence. Depending on what you are working on, and your own preferences, you will want to alter the default settings for many of these options.

Canceling an Editing Session

Finally, to exit **vi** without saving the file *practice*, enter:

```
:q!
```

and press ⟨Return⟩. This cancels all the changes you have made to *practice* and, since it is a new file, deletes it. The prompt appears. If *practice* had already existed before this editing session, the changes you made would be disregarded, but the file would still exist.

This completes the demonstration. You have learned how to get in and out of **vi**, insert and delete text, move the cursor around, make searches and replacements, how to execute line-oriented commands, copy text from other files, and cancel an editing session.

There are many more commands to learn, but the fundamentals of using **vi** have been covered. The following sections will give you more detailed information about these commands and about other **vi** commands and features.

Editing Tasks

The following sections explain how to perform common editing tasks. By following the instructions in each section you will be able to complete each task described. Features that are needed in several tasks are described each time they are used, so some information is repeated.

How to Enter the Editor

There are several ways to begin editing, depending on what you are planning to do. This section describes how to start, or "invoke" the editor with one filename. To invoke **vi** on a series of files, see the section "Editing a Series of Files."

With a Filename

The most common way to enter **vi** is to enter the command **vi** and the name of the file you wish to edit:

vi *filename*

If *filename* does not already exist, a new, empty file is created.

At a Particular Line

You can also enter the editor at a particular place in a file. For example, if you wish to start editing a file at line 100, enter:

vi +100 *filename*

The cursor is placed at line 100 of *filename*.

At a Particular Word

If you wish to begin editing at the first occurrence of a particular word, enter:

vi *+/word filename*

The cursor is placed at the first occurrence of *word*. For example, to begin editing the file *temp* at the the first occurrence of *contain*, enter:

```
vi +/contain temp
```


Moving the Cursor

The cursor movement keys allow you to move the cursor around in a file. Cursor movement can only be done in Command mode.

Moving the Cursor by Characters: h, l, f, F, t, T, ⟨Space⟩, ⟨Bksp⟩

The ⟨Space⟩ bar and the **l** key move the cursor forward a specified number of characters. The ⟨Bksp⟩ key and the **h** key move it backward a specified number of characters. If no number is specified, the cursor moves one character. For example, to move backward four characters, enter:

```
4h
```

You can also move the cursor to a designated character on the current line. **F** moves the cursor back to the specified character, **f** moves it forward. The cursor rests on the specified character. For example, to move the cursor backward to the nearest *p* on the current line, enter:

```
Fp
```

To move the cursor forward to the nearest *p*, enter:

```
fp
```

The **t** and **T** keys work the same way as **f** and **F**, but place the cursor immediately before the specified character. For example, to move the cursor back to the space next to the nearest *p* in the current line, enter:

```
Tp
```

If the *p* were in the word *telephone*, the cursor would sit on the *h*.

The cursor always remains on the same line when you use these commands. If you specify a number greater than the number of characters on the line, the cursor does not move beyond the beginning or end of that line.

Moving the Cursor by Lines: j, k

The **j** key moves the cursor down a specified number of lines, and the **k** key moves it up. If no number is specified, the cursor moves one line. For example, to move down three lines, enter:

```
3j
```


Moving the Cursor by Words: w, W, b, B, e, E

The **w** key moves the cursor forward to the beginning of the specified number of words. Punctuation and nonalphabetic characters (such as !@#$%^&*()_+{}[]~|\'<>/) are considered words, so if a word is followed by a comma the cursor will count the comma in the specified number.

For example, your cursor rests on the first letter of this sentence:

```
No, I didn't know he had returned.
```

If you press:

```
6w
```

the cursor stops on the *k* in *know*.

W works the same way as **w**, but includes punctuation and nonalphabetic characters as part of the word. Using the above example, if you press:

```
6W
```

the cursor stops on the *r* in *returned*; the comma and the apostrophe are included in their adjacent words.

The **e** and **E** keys move the cursor forward to the end of a specified number of words. The cursor is placed on the last letter of the word. The **e** command counts punctuation and nonalphabetic characters as separate words; **E** does not.

B and **b** move the cursor back to the beginning of a specified number of words. The cursor is placed on the first letter of the word. The **b** command counts punctuation and nonalphabetic characters as separate words; **B** does not. Using the above example, if the cursor is on the *r* in *returned*, enter:

```
4b
```

and the cursor moves to the *t* in *didn't*.

Enter:

```
4B
```

and the cursor moves to the first *d* in *didn't*.

The **w**, **W**, **b** and **B** commands will move the cursor to the next line if that is where the designated word is, unless the current line ends in a space.

Moving the Cursor by Lines

Forward: j, ⟨Ctrl⟩n, +, ⟨Return⟩, LINEFEED, $

The ⟨Return⟩, **LINEFEED** and + keys move the cursor forward a specified number of lines, placing the cursor on the first character. For example, to move the cursor forward six lines, enter:

```
6+
```

The **j** and ⟨Ctrl⟩n keys move the cursor forward a specified number of lines. The cursor remains in the same place on the line, unless there is no character in that place, in which case it moves to the last character on the line. For example, in the following two lines if the cursor is resting on the *e* in *characters*, pressing **j** moves it to the period at the end of the second line:

```
Lines contain characters.
Text contains lines.
```

The dollar sign($) moves the cursor to the end of a specified number of lines. For example, to move the cursor to the last character of the line four lines down from the current line, enter:

```
4$
```

Backward: k, ⟨Ctrl⟩p

⟨Ctrl⟩p and **k** move the cursor backward a specified number of lines, keeping it on the same place on the line. For example, to move the cursor backward four lines from the current line, enter:

```
4k
```

Moving the Cursor on the Screen: H, M, L

The **H**, **M** and **L** keys move the cursor to the beginning of the top, middle and bottom lines of the screen, respectively.

Moving Around in a File: Scrolling

The following commands move the file so different parts can be displayed on the screen. The cursor is placed on the first letter of the last line scrolled.

Scrolling Up Part of the Screen: ⟨Ctrl⟩u

⟨Ctrl⟩u scrolls up one-half screen.

Scrolling Up the Full Screen: ⟨Ctrl⟩b

⟨Ctrl⟩b scrolls up a full screen.

Scrolling Down Part of the Screen: ⟨Ctrl⟩d

⟨Ctrl⟩d scrolls down one-half screen.

Scrolling Down a Full Screen: ⟨Ctrl⟩f

⟨Ctrl⟩f scrolls down a full screen.

Placing a Line at the Top of the Screen: z

To scroll the current line to the top of the screen, press:

```
z
```

then press ⟨Return⟩. To place a specific line at the top of the screen, precede the *z* with the line number, as in

```
33z
```

Press ⟨Return⟩, and line 33 scrolls to the top of the screen. For information on how to display line numbers, see the section "Displaying Line Numbers: number."

Inserting Text Before the Cursor: i and I

You can begin inserting text before the cursor anywhere on a line, or at the beginning of a line. In order to insert text into a file, you must be in Insert mode. To enter Insert mode press:

i

The "i" does not appear on the screen. Any text typed after the "i" becomes part of the file you are editing. To leave Insert mode and reenter Command mode, press ⟨Esc⟩. For more explanation of modes in **vi**, see the section "Inserting Text."

Anywhere on a Line: i

To insert text before the cursor, use the **i** command. Press the *i* key to enter Insert mode (the "i" does not appear on your screen), then begin entering your text. To leave Insert mode and reenter Command mode, press ⟨Esc⟩.

At the Beginning of the Line: I

Using an uppercase "I" to enter Insert mode also moves the cursor to the beginning of the current line. It is used to start an insertion at the beginning of the current line.

Appending After the Cursor: a and A

You can begin appending text after the cursor anywhere on a line, or at the end of a line. Press ⟨Esc⟩ to leave Insert mode and reenter Command mode.

Anywhere on a Line: a

To append text after the cursor, use the **a** command. Press the a key to enter Insert mode (the "a" does not appear on your screen), then begin entering your text. Press ⟨Esc⟩ to leave Insert mode and reenter Command mode.

At the end of a Line: A

Using an uppercase "A" to enter Insert mode also moves the cursor to the end of the current line. It is useful for appending text at the end of the current line.

Correcting Typing Mistakes

If you make a mistake while you are typing, the simplest way to correct it is with the ⟨Bksp⟩ key. Backspace across the line until you have backspaced over the mistake, then retype the line. You can only do this, however, if the cursor is on the same line as the error. See the sections "Deleting a Character: x and X" through "Deleting an Entire Insertion" for other ways to correct typing mistakes.

Opening a New Line

To open a new line above the cursor, press **O**. To open a new line below the cursor, press **o**. Both commands place you in Insert mode, and you may begin entering immediately. Press ⟨Esc⟩ to leave Insert mode and reenter Command mode.

You may also use the ⟨Return⟩ key to open new lines above and below the cursor. To open a line above the cursor, move the cursor to the beginning of the line, press **i** to enter Insert mode, then press ⟨Return⟩. (For information on how to move the cursor, see the section "Moving the Cursor.") To open a line below the cursor, move the cursor to the end of the current line, press **i** to enter Insert mode, then press ⟨Return⟩.

Repeating the Last Insertion

⟨Ctrl⟩@ repeats the last insertion. Press **i** to enter Insert mode, then press ⟨Ctrl⟩@.

⟨Ctrl⟩@ only repeats insertions of 128 characters or less. If more than 128 characters were inserted, ⟨Ctrl⟩@ does nothing.

For other methods of repeating an insertion, see the sections "Repeating the Last Insertion," "Inserting Text From Other Files," and "Repeating a Command."

Inserting Text From Other Files

To insert the contents of another file into the file you are currently editing, use the Read (r) command. Move the cursor to the line immediately *above* the place you want the new material to appear, then enter:

:r *filename*

where *filename* is the file containing the material to be inserted, and press ⟨Return⟩. The text of *filename* appears on the line below the cursor, and the cursor moves to the first character of the new text. This text is a copy; the original *filename* still exists.

Inserting selected lines from another file is more complicated. The selected lines are copied from the original file into a temporary holding place called a "buffer", then inserted into the new file.

1. To select the lines to be copied, save your original file with the Write (:w) command , but do not exit vi.

2. Enter:

   :e *filename*

   where *filename* is the file that contains the text you want to copy, and press ⟨Return⟩.

3. Move the cursor to the first line you wish to select.

4. Enter:

   mk

   This "marks" the first line of text to be copied into the new file with the letter "k".

5. Move the cursor to the last line of the selected text. Enter:

   "ay'k

   The lines from your first "mark" to the cursor are placed, or "yanked" into buffer *a*. They will remain in buffer *a* until you replace them with other lines, or until you exit the editor.

6. Enter:

   :e#

   to return to your previous file. (For more information about this command, see the section "Editing a New File Without Leaving the Editor.") Move the cursor to the line above the place you want the new text to appear, then enter:

   "ap

This "puts" a copy of the yanked lines into the file, and the cursor is placed on the first letter of this new text. The buffer still contains the original yanked lines.

You can have 26 buffers named *a, b, c,* up to and including *z*. To name and select different buffers, replace the *a* in the above examples with whatever letter you wish.

You may also delete text into a buffer, then insert it in another place. For information on this type of deletion and insertion, see the section "Moving Text."

Copying Lines From Elsewhere in the File

To copy lines from one place in a file to another place in the same file, use the Copy (**co**) command.

co is a line-oriented command, and to use it you must know the line numbers of the text to be copied and its destination. To find out the number of the current line enter:

:nu

and press ⟨Return⟩. The line number and the text of that line are displayed on the status line. To find out the destination line number, move the cursor to the line above where you want the copied text to appear and repeat the **:nu** command. You can also make line numbers appear throughout the file with the **linenumber** option. For information on how to set this option, see the section "Displaying Line Numbers: number." The following example uses the **number** option to display line numbers in a file.

```
1 [F]iles contain text.
2 Text contains lines.
3 Lines contain characters.
4 Characters form words.
5 Words form text.
~
~
~
~
~
```

Using the above example, to copy lines 3 and 4 and put them between lines 1 and 2, enter:

```
:3,4 co 1
```

The result is:

```
1 Files contain text.
2 Lines contain characters.
3 [C]haracters form words.
4 Text contains lines.
5 Lines contain characters.
6 Characters form words.
7 Words form text.
~
~
~
```

If you have text that is to be inserted several times in different places, you can save it in a temporary storage area, called a "buffer", and insert it whenever it is needed. For example, to repeat the first line of the following text after the last line:

```
[F]iles contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

1. Move the cursor over the *F* in *Files*. Enter the following line, which will not be echoed on your screen:

   ```
   "ayy
   ```

   This "yanks" the first line into buffer *a*. Move the cursor over the *W* in *Words*.

2. Enter the following line:

   "ap

   This "puts" a copy of the yanked line into the file, and the cursor is placed on the first letter of this new text. The buffer still contains the original yanked line.

Your screen looks like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
[F]iles contain text.
~
~
~
~
```

If you wish to "yank" several consecutive lines, indicate the number of lines you wish to yank after the name of the buffer. For example, to place three lines from the above text in buffer *a*, enter:

"a3yy

You can also use "yank" to copy parts of a line. For example, to copy the words *Files contain*, enter:

2yw

This yanks the next two words, including the word on which you place the cursor. To yank the next ten characters, enter:

10yl

l indicates cursor motion to the right. To yank to the end of the line you are on, from where you are now, enter:

y$

Inserting Control Characters into Text

Many control characters have special meaning in **vi**, even when typed in Insert mode. To remove their special significance, press ⟨Ctrl⟩v before typing the control character. Note that ⟨Ctrl⟩j, ⟨Ctrl⟩q, and ⟨Ctrl⟩s cannot be inserted as text. ⟨Ctrl⟩j is a newline character. ⟨Ctrl⟩q and ⟨Ctrl⟩s are meaningful to the operating system, and are trapped by it before they are interpreted by **vi**.

Joining and Breaking Lines

To join two lines press:

J

while the cursor is on the first of the two lines you wish to join.

To break one line into two lines, position the cursor on the space preceding the first letter of what will be the second line, press:

r

then press ⟨Return⟩.

Deleting a Character: x and X

The **x** and **X** commands delete a specified number of characters. The **x** command deletes the character above the cursor; the **X** command deletes the character immediately before the cursor. If no number is given, one character is deleted. For example, to delete three characters following the cursor (including the character above the cursor), enter:

3x

To delete three characters preceding the cursor, enter:

3X

Deleting a Word: dw

The **dw** command deletes a specified number of words. If no number is given, one word is deleted. A word is interpreted as numbers and letters separated by whitespace. When a word is deleted, the space after it is also deleted. For example, to delete three words, enter:

```
3dw
```

Deleting a Line: D and dd

The **D** command deletes all text following the cursor on that line, including the character the cursor is resting on. The **dd** command deletes a specified number of lines and closes up the space. If no number is given, only the current line is deleted. For example, to delete three lines, enter:

```
3dd
```

Another way to delete several lines is to use a line-oriented command. To use this command it helps to know the line numbers of the text you wish to delete. For information on how to display line numbers, see the section "Displaying Line Numbers: number."

For example, to delete lines 200 through 250, enter:

```
:200,250d
```

Press ⟨Return⟩.

When the command finishes, the message:

```
50 lines
```

appears on the **vi** status line, indicating how many lines were deleted.

It is possible to remove lines without displaying line numbers using shorthand "addresses". For example, to remove all lines from the current line (the line the cursor rests on) to the end of the file, enter:

```
:.,$d
```

The dot (.) represents the current line, and the dollar sign stands for the last line in the file. To delete the current line and 3 lines following it, enter:

```
:.,+3d
```

To delete the current line and 3 lines preceding it, enter:

```
:.,-3d
```

For more information on using addresses in line-oriented commands, see **vi**(C) in the *XENIX Reference*.

Deleting an Entire Insertion

If you wish to delete all of the text you just entered, press ⟨Ctrl⟩u while you are in Insert mode. The cursor returns to the beginning of the insertion. The text of the original insertion is still displayed, and any text you enter replaces it. When you press ⟨Esc⟩, any text remaining from the original insertion disappears.

Deleting and Replacing Text

Several **vi** commands combine removing characters and entering Insert mode. The following sections explain how to use these commands.

Overstriking: r and R

The **r** command replaces the character under the cursor with the next character entered. To replace the character under the cursor with a "b", for example, enter:

```
rb
```

If a number is given before **r**, that number of characters is replaced with the next character entered. For example, to replace the character above the cursor, plus the next three characters, with the letter "b", enter:

```
4rb
```

Note that you now have four "b"s in a row.

The **R** command replaces as many characters as you enter. To end the replacement, press ⟨Esc⟩. For example, to replace the second line in the following text with "Spelling is important.":

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Move the cursor over the *T* in *Text*. Press R, then enter:

```
Spelling is important.
```

Press ⟨Esc⟩ to end the replacement. If you make a mistake, use the ⟨Bksp⟩ key to correct it. Your screen should now look like this:

```
Files contain text.
Spelling is important[.]
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Substituting: s and S

The **s** command replaces a specified number of characters, beginning with the character under the cursor, with text you enter. For example, to substitute "xyz" for the cursor and two characters following it, enter:

```
3sxyz
```


The **S** command deletes a specified number of lines and replaces them with text you enter. You may enter as many new lines of text as you wish; **S** affects only how many lines are deleted. If no number is given, one line is deleted. For example, to delete four lines, including the current line, enter:

```
4S
```

This differs from the **R** command. The **S** command deletes the entire current line; the **R** command deletes text from the cursor onward.

Replacing a Word: cw

The **cw** command replaces a word with text you enter. For example, to replace the word "bear" with the word "fox", move the cursor over the "b" in "bear". Press:

```
cw
```

A dollar sign appears over the "r" in *bear*, marking the end of the text that is being replaced. Enter:

```
fox
```

and press ⟨Esc⟩. The rest of "bear" disappears and only "fox" remains.

Replacing the Rest of a Line: C

The C command replaces text from the cursor to the end of the line. For example, to replace the text of the sentence:

Who's afraid of the big bad wolf?

from *big* to the end, move the cursor over the *b* in *big* and press:

C

A dollar sign (*$*) replaces the question mark (*?*) at the end of the line. Enter the following:

little lamb?

Press ⟨Esc⟩. The remaining text from the original sentence disappears.

Replacing a Whole Line: cc

The **cc** command deletes a specified number of lines, regardless of the location of the cursor, and replaces them with text you enter. If no number is given, the current line is deleted.

Replacing a Particular Word on a Line

If a word occurs several times on one line, it is often convenient to use a line-oriented command to replace it. For example, to replace the word *removing* with "deleting" in the following sentence:

In vi, removing a line is as easy as removing a letter.

Make sure the cursor is at the beginning of that line, and enter:

:s/removing/deleting/g

Press ⟨Return⟩. This line-oriented command means "Substitute (s) for the word *removing* the word *deleting*, everywhere it occurs on the current line (g)". If you don't include a g at the end, only the first occurrence of *removing* is changed.

For more information on using line-oriented commands to replace text, see the section "Searching and Replacing."

Moving Text

To move a block of text from one place in a file to another, you can use the line-oriented **m** command. You must know the line numbers of your file to use this command. The **number** option displays line numbers. To set this option, press ⟨Esc⟩ to make sure you are in Command mode, then enter:

```
set number
```

Line numbers will appear to the left of your text. For more information on setting the **number** option, see the section "Displaying Line Numbers: number."

The following example uses the **number** option. For other ways to display line numbers, see the section "Finding Out What Line You're On."

```
1 [F]iles contain text.
2 Text contains lines.
3 Lines contain characters.
4 Characters form words.
5 Words form text.
~
~
~
~
~
```

To insert lines 2 and 3 between lines 4 and 5, enter:

```
:2,3m4
```

Your screen should look like this:

```
1 Files contain text.
2 Characters form words.
3 Text contains lines.
4 Lines contain characters.
5 [W]ords form text.
~
~
~
~
~
```

To place line 5 after line 2, enter:

```
:5m2
```

After moving, your screen should look like this:

```
1 Files contain text.
2 Characters form words.
3 [W]ords form text.
4 Text contains lines.
5 Lines contain characters.
~
~
~
~
~
```

To make line 4 the first line in the file, enter:

```
:4m0
```

Your screen should look like this:

```
1 [T]ext contains lines.
2 Files contain text.
3 Characters form words.
4 Words form text.
5 Lines contain characters.
~
~
~
~
~
```


You can also delete text into a temporary storage place, called a "buffer," and insert it wherever you wish. When text is deleted it is placed in a "delete buffer." There are nine "delete buffers."

The first buffer always contains the most recent deletion. In other words, the first deletion in a given editing session goes into buffer 1. The second deletion also goes into buffer 1, and pushes the contents of the old buffer 1 into buffer 2. The third deletion goes into buffer 1, pushing the contents of buffer 2 into buffer 3, and the contents of buffer 1 into buffer 2. When buffer 9 has been used, the next deletion pushes the current text of buffer 9 off the stack and it disappears.

Text remains in the delete buffers until it is pushed off the stack, or until you quit the editor, so it is possible to delete text from one file, change files without leaving the editor, and place the deleted text in another file.

Delete buffers are particularly useful when you wish to remove text, store it, and put it somewhere else. Using the following text as an example:

```
[F]iles contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Delete the first line by entering:

```
dd
```

Delete the third line the same way. Now move the cursor to the last line in the example and press:

```
"1p
```

The line from the *second* deletion appears:

```
Text contains lines.
Characters form words.
Words form text.
[L]ines contain characters.
~
~
~
~
~
~
```

Now enter:

```
"2p
```

The line from the *first* deletion appears:

```
Text contains lines.
Characters form words.
Words form text.
Lines contain characters.
[F]iles contain text.
~
~
~
~
~
```

Inserting text from a delete buffer does not remove the text from the buffer. Since the text remains in a buffer until it is either pushed off the stack or until you quit the editor, you may use it as many times as you wish.

It is also possible to place text in named buffers. For information on how to create named buffers, see the section "Inserting Text From Other Files."

Searching: / and ?

You can search forward and backward for patterns in vi. To search forward, press the slash (/) key. The slash appears on the status line. Enter the characters you wish to search for. Press ⟨Return⟩. If the specified pattern exists, the cursor will move to the first character of the pattern.

For example, to search forward in the file for the word "account", enter:

```
/account
```

Press ⟨Return⟩. The cursor is placed on the first character of the pattern. To place the cursor at the beginning of the line above "account", for example, enter:

```
/account/-
```

To place the cursor at the beginning of the line two lines above the line that contains "account", enter:

```
/account/-2
```

To place the cursor two lines below "account", enter:

```
/account/+2
```

To search backward through a file, use ? instead of / to start the search. For example, to find all occurrences of "account" above the cursor, enter:

```
?account
```

To search for a pattern containing any of the special characters (. * \ [ ] ~ $ and ^), each special character must be preceded by a backslash. For example, to find the pattern "U.S.A.", enter:

```
/U\.S\.A\./
```

You can continue to search for a pattern by pressing:

n

after each search. The pattern is unaffected by intervening **vi** commands, and you can use **n** to search for the pattern until you enter a new pattern or quit the editor.

vi searches for exactly what you enter. If the pattern you are searching for contains an uppercase letter (for example, if it appears at the beginning of a sentence), **vi** ignores it. To disregard case in a search command, you can set the ignorecase option:

:set ignorecase

By default, searches "wrap around" the file. That is, if a search starts in the middle of a file, when **vi** reaches the end of the file it will "wrap around" to the beginning, and continue until it returns to where the search began. Searches will be completed faster if you specify forward or backward searches, depending on where you think the pattern is.

If you do not want searches to wrap around the file, you can change the "wrapscan" option setting. Enter:

:set nowrapscan

and press ⟨Return⟩ to prevent searches from wrapping. For more information about setting options, see the section "Setting Up Your Environment."

Searching and Replacing

The search and replace commands allow you to perform complex changes to a file in a single command. Learning how to use these commands is a must for the serious user of **vi**.

The syntax of a search and replace command is:

g/*pattern1*/s/[*pattern2*]/[*options*]

Brackets indicate optional parts of the command line. The g tells the computer to execute the replacement on every line in the file. Otherwise the replacement would occur only on the current line. The *options* are explained in the following sections.

To explain these commands we will use the example file from the demonstration run:


```
[F]iles contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Replacing a Word

To replace the word "contain" with the word "are" throughout the file, enter the following command:

```
:g/contain /s//are /g
```

This command says "On each line of the file (g), find *contain* and substitute for that word (s//) the word *are*, everywhere it occurs on that line (the second g)". Note that a space is included in the search pattern for *contain*; without the space *contains* would also be replaced.

After the command executes your screen should look like this:

```
[F]iles are text.
Text contains lines.
Lines are characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Printing all Replacements

To replace "contain" with "are" throughout the file, and print every line changed, use the **p** option:

```
:g/contain /s//are /gp
```

Press ⟨Return⟩. After the command executes, each line in which "contain" was replaced by "are" is printed on the lower part of the screen. To remove these lines, redraw the screen by pressing ⟨Ctrl⟩r.

Choosing a Replacement

Sometimes you may not want to replace every instance of a given pattern. The **c** option displays every occurrence of *pattern* and waits for you to confirm that you want to make the substitution. If you press **y** the substitution takes place; if you press ⟨Return⟩ the next instance of *pattern* is displayed.

To run this command on the example file, enter:

```
:g/contain/s//are/gc
```

Press ⟨Return⟩. The first instance of "contain" appears on the status line:

```
Files contain text.
```

Press **y** , then ⟨Return⟩. The next occurrence of *contain* appears.

Pattern Matching

Search commands often require, in addition to the characters you want to find, a context in which you want to find them. For example, you may want to locate every occurrence of a word at the beginning of a line. **vi** provides several special characters that specify particular contexts.

Matching the Beginning of a Line

When a caret(^) is placed at the beginning of a pattern, only patterns found at the beginning of a line are matched. For example, the following search pattern only finds "text" when it occurs as the first word on a line:

```
/^text/
```


To search for a caret that appears as text you must precede it with a backslash (\).

Matching the End of a Line

When a dollar sign ($) is placed at the end of a pattern, only patterns found at the end of a line are matched. For example, the following search pattern only finds "text" when it occurs as the last word on a line:

```
/text$/
```

To search for a dollar sign that appears as text you must precede it with a backslash (\).

Matching Any Single Character

When used in a search pattern, the period (.) matches any single character except the newline character. For example, to find all words that end with "ed", use the following pattern:

```
/.ed  /
```

Note the space between the *d* and the backslash.

To search for a period in the text, you must precede it with a backslash (\).

Matching a Range of Characters

A set of characters enclosed in square brackets matches any single character in the range designated. For example, the search pattern:

```
/[a-z]/
```

finds any lowercase letter. The search pattern:

```
/[aA]pple/
```

finds all occurrences of "apple" and "Apple".

To search for a bracket that appears as text, you must precede it with a backslash (\).

Matching Exceptions

A caret (^) at the beginning of *string* matches every character *except* those specified in *string*. For example the search pattern:

```
[^a-z]
```

finds anything but a lowercase letter or a newline.

Matching the Special Characters

To place a caret, hyphen or square bracket in a search pattern, precede it with a backslash. To search for a caret, for example, enter:

```
/\^/
```

If you need to search for many patterns that contain special characters, you can reset the **magic** option. To do this, enter:

```
:set nomagic
```

This removes the special meaning from the characters ., \, $, [and]. You can include them in search and replace commands without a preceding backslash. Note that the special meaning cannot be removed from the special characters star (*) and caret (^); these must always be preceded by a backslash in searches.

To restore *magic*, enter:

```
:set magic
```

For more information about setting options, see the "Setting Up Your Environment" section.

Undoing a Command: u

Any editing command can be reversed with the Undo (**u**) command. The Undo command works on both screen-oriented and line-oriented commands. For example, if you have deleted a line and then decide you wish to keep it, press *u* and the line will reappear.

Use the following line as an example:

```
[T]ext contains lines.
~
~
~
~
~
~
~
~
~
```

Place the cursor over the "c" in "contains", then delete the word with the **dw** command. Your screen should look like this:

```
Text [l]ines.
~
~
~
~
~
~
~
~
~
```

Press **u** to undo the **dw** command. *contains* reappears:

```
Text [c]ontains lines.
~
~
~
~
~
~
~
~
~
```

If you press **u** again, "contains" is deleted again:

```
Text [l]ines.
~
~
~
~
~
~
~
~
~
```


It is important to remember that **u** only undoes the *last* command. For example, if you make a global search and replace, then delete a few characters with the **x** command, pressing **u** will undo the deletions but not the global search and replace.

Repeating a Command: .

Any screen-oriented **vi** command can be repeated with the Repeat (.) command. For example, if you have deleted two words by entering:

```
2dw
```

you may repeat this command as many times as you wish by pressing the period key (.). Cursor movement does not affect the Repeat command, so you may repeat a command as many times and in as many places in a file as you wish.

The Repeat command only repeats the last **vi** command. Careful planning can save time and effort. For example, if you want to replace a word that occurs several times in a file (and for some reason you do not wish to use a global command), use the **cw** command instead of deleting the word with the **dw** command, then inserting new text with the **i** command. By using the **cw** command you can repeat the replacement with the dot (.) command. If you delete the word, then insert new text, dot only repeats the replacement.

Leaving the Editor

There are several ways to exit the editor and save any changes you may have made to the file. One way is to enter:

```
:x
```


and press ⟨Return⟩. This command replaces the old copy of the file with the new one you have just edited, quits the editor, and returns you to the XENIX shell. Similarly, if you enter:

```
ZZ
```

the same thing happens, except the old copy file is written out *only* if you have made any changes. Note that the **ZZ** command is *not* preceded by a colon, and is not echoed on the screen.

To leave the editor without saving any changes you have made to the file, enter:

```
:q!
```

The exclamation point tells **vi** to quit unconditionally. If you leave out the exclamation point:

```
:q
```

vi will not let you quit. You will see the error message:

```
No write since last change (:quit! overrides)
```

This message tells you to use **:q!** if you really want to leave the editor without saving your file.

Saving a File Without Leaving the Editor

There are many occasions when you must save a file without leaving the editor, such as when starting a new shell, or moving to another file. Before you can perform these tasks you must first save the current file with the Write (**:w**) command:

```
:w
```

You do not need to enter the name of the file; **vi** remembers the name you used when you invoked the editor. If you invoked **vi** without a filename, you may name the file by entering:

```
:w filename
```

where *filename* is the name of the new file.

Editing a Series of Files

Entering and leaving **vi** for each new file takes time, particularly on a heavily used system, or when you are editing large files. If you have many files to edit in one session, you can invoke **vi** with more than one filename, and thus edit more than one file without leaving the editor, as in:

```
vi file1 file2 file3 file4 file5 file6
```

But entering many filenames is tedious, and you may make a mistake. If you mistype a filename, you must either backspace over to mistake and reenter the line, or kill the whole line and reenter it. It is more convenient to invoke **vi** using the special characters as abbreviations.

To invoke **vi** on the above files without typing each name, enter:

```
vi file*
```

This invokes **vi** on all files that begin with the letters "file". You can plan your filenames to save time in later editing. For example, if you are writing a document that consists of many files, it would be wise to give each file the same filename extension, such as ".s". Then you can invoke **vi** on the entire document:

vi *.s

You can also invoke **vi** on a selected range of files:

vi [3-5]*.s

or

vi [a-h]*

To invoke **vi** on all files that are five letters long, and have any extension:

vi ?????.*

For more information on using special characters, see "Naming Conventions" in the "Basic Concepts" chapter of the *XENIX Tutorial.*

When you invoke **vi** with more than one filename, you will see the following message when the first file is displayed on the screen:

x files to edit

After you have finished editing a file, save it with the Write (**:w**) command, then go to the next file with the Next (**:n**) command:

:n

The next file appears, ready to edit. It is not necessary to specify a filename; the files are invoked in alphabetical (or numerical, if the filenames begin with numbers) order.

If you forget what files you are editing, enter:

:args

The list of files appears on the status line. The current file is enclosed in square brackets.

To edit a file out of order, such as *file4* after *file2*, enter:

```
:e file4
```

instead of using the (**:n**) command. If you enter:

```
:n
```

after you finish editing *file4*, you will go back to *file3*.

If you wish to start again from the beginning of the list, enter:

```
:rew
```

To discard the changes you made and start again at the beginning, enter:

```
:rew!
```

Editing a New File Without Leaving the Editor

You can start editing another file anywhere on a XENIX system without leaving **vi**. This saves time when you wish to edit several files in one session that are in different directories, or even in the same directory. For example, if you have finished editing */usr/joe/memo* and you wish to edit */usr/mary/letter*, first save the file *memo* with the Write (**:w**) command then enter:

```
:e /usr/mary/letter
```

/usr/mary/letter appears on your screen just as though you had left **vi**.

Note

You *must* write out your file with the Write (:w) command to save the changes you have made. If you try to edit a second file without writing out the first file, the message "No write since last change (:e! overrides)" appears. If you use **:e!** all your changes to the first file are discarded.

If you want to switch back and forth between two files, vi remembers the name of the last file edited. Using the above example, if you wish to go back and edit the file */usr/joe/memo* after you have finished with */usr/mary/letter*, enter:

```
:e#
```

The cursor is positioned in the same location it was when you first saved */usr/joe/memo*.

Leaving the Editor Temporarily: Shell Escapes

You can execute any XENIX command from within vi using the shell Escape (!) command. For example, if you wish to find out the date and time, enter:

```
:!date
```

The exclamation point sends the remainder of the line to the shell to be executed, and the date and time appear on the vi status line. You can use the ! to perform any XENIX command. To send mail to joe without leaving the editor, enter:

```
:!mail joe
```

Type your message and send it. (For more information about the XENIX mail system, see the "mail" chapter.) After you send it, the message

```
[Press return to continue]
```

appears. Press ⟨Return⟩ to continue editing.

If you want to perform several XENIX commands before returning to the editor, you can invoke a new shell:

```
:!sh
```

The XENIX prompt appears. You may execute as many commands as you like. Press ⟨Ctrl⟩d to terminate the new shell and return to your file.

If you have not written out your file before a shell escape, you will see the message:

```
[No write since last change]
```

It is a good idea to save your file with the Write (**:w**) command before executing an escape, just in case something goes wrong. However, once you become an experienced **vi** user, you may wish to turn off this message. To turn off the "No write" message, reset the **warn** option, as follows:

```
:set nowarn
```

For more information about setting options in **vi**, see the section "Setting Up Your Environment."

Performing a Series of Line-Oriented Commands: Q

If you have several line-oriented commands to perform, you can place yourself temporarily in Line-oriented mode by entering:

```
Q
```

while you are in Command mode. A colon prompt appears on the status line.

Commands executed in this mode cannot be undone with the **u** command, nor do they appear on the screen until you re-enter Normal **vi** mode. To re-enter Normal **vi** mode, enter:

```
vi
```

Finding Out What File You're In

If you forget what file you are editing, press ⟨Ctrl⟩g while you are in Command mode. A line similar to the following appears appears on the status line:

```
"memo" [Modified] line 12 of 100 --12%--
```


From left to right, the following information is displayed:

- The name of the file
- Whether or not the file has been modified
- The line number the cursor is on
- How many lines there are in the file
- Your location in the file (expressed as a percentage)

This command is also useful when you need to know the line number of the current line for a line-oriented command.

The same information can be obtained by entering:

:file

or

:f

Finding Out What Line You're On

To find out what line of the file you are on, enter:

:nu

and press ⟨Return⟩. This command displays the current line number and the text of the line.

To display line numbers for the entire file, see the section "Displaying Line Numbers: number."

Solving Common Problems

The following is a list of common problems that you may encounter when using **vi**, along with the probable solution.

- *I don't know which mode I'm in.*

  Press ⟨Esc⟩ until the bell rings. When the bell rings you are in Command mode.

- *I can't get out of a subshell.*

  Press ⟨Ctrl⟩d to exit any subshell. If you have created more than one subshell (not a good idea, usually), keep pressing ⟨Ctrl⟩d until you see the message:

  ```
  [Press return to continue]
  ```

- *I made an inadvertent deletion (or insertion).*

  Press **u** to undo the last Delete or Insert command.

- *There are extra characters on my screen.*

  Press ⟨Ctrl⟩l to redraw the screen.

- *When I type, nothing happens.*

  vi has crashed and you are now in the shell with your terminal characteristics set incorrectly. To reset the keyboard, slowly enter:

  stty sane

  then press ⟨Ctrl⟩j or **LINEFEED**. Pressing ⟨Ctrl⟩j instead of ⟨Return⟩ is important here, since it is quite possible that the ⟨Return⟩ key will not work as a newline character. To make sure that other terminal characteristics have not been altered, log off, turn your terminal off, turn your terminal back on, and then log back in. This should guarantee that your terminal's characteristics are back to normal. This procedure may vary somewhat depending on the terminal.

- *The system crashed while I was editing.*

  Normally, **vi** will inform you (by sending you mail) that your file has been saved before a crash. The file can be recovered by entering:

  vi -r *filename*

  If **vi** was unable to save the file before the crash, it is irretrievably lost.

- *I keep getting a colon on the status line when I press* **⟨Return⟩**

  You are in line-oriented Command mode. Enter:

  vi

  to return to normal **vi** Command mode.

- *I get the error message "Unknown terminal type [Using open mode]" when I invoke vi.*

  Your terminal type is not set correctly. To leave Open mode, press ⟨Esc⟩, then enter:

  :wq

  and press ⟨Return⟩. Turn to the section "Setting the Terminal Type" for information on how to set your terminal type correctly.

Setting Up Your Environment

There are a number of options that can be set that affect your terminal type, how files and error messages are displayed on your screen, and how searches are performed. These options can be set with the **set** command while you are editing, they can be defined with the **EXINIT** environment variable (see the *environ*(M) manual page), or they can be placed in the **vi** *.exrc* startup file (see "Customizing Your Environment: The .exrc File").

You can also define mappings and abbreviations to reduce repetitive tasks with the **map** and **abbr** commands while you are editing, with **EXINIT**, or in the *.exrc* file.

The following sections describe how to set some commonly used options and how to create mappings and abbreviations. There is a complete list of options in **vi**(C) in the *XENIX Reference*.

Setting the Terminal Type

Before you can use **vi**, you must set the terminal type, if this has not already been done for you, by defining the TERM variable in your *.profile* or *.login* file. The TERM variable is a number that tells the operating system what type of terminal you are using. To determine this number you must find out what type of terminal you are using. Then look up this type in **terminals**(M) in the *XENIX Reference*. If you cannot find your terminal type or its number, consult your System Administrator.

For these examples, we will suppose that you are using an HP 2621 terminal. For the HP 2621, the TERM variable is "2621". How you define this variable depends on which shell you are using. You can usually determine which shell you are using by examining the prompt character. The Bourne shell prompts with a dollar sign ($); the C-shell prompts with a percent sign (%).

Setting the TERM Variable: The Bourne or Korn Shell

To set your terminal type to 2621 place the following commands in the file *.profile*:

```
TERM=2621
export TERM
```

Setting the TERM Variable: The C Shell

To set your terminal type to 2621 for the C shell, place the following command in the file *.login*:

```
setenv TERM 2621
```


Setting Options: The set Command

The **set** command is used to display option settings and to set options.

Listing the Available Options

To get a list of the options available to you and how they are set, enter:

```
:set all
```

Your display should look similar to this:

```
noautoindent    open                        noslowopen
autoprint       nooptimize                  tabstop=8
noautowrite     paragraphs=IPLPPPQPP LIbp   taglength=0
nobeautify      noprompt                    ttytype=h19
directory=/tmp  noreadonly                  term=h19
noerrorbells    redraw                      noterse
hardtabs=8      report=5                    warn
noignorecase    scroll=4                    window=8
nolisp          sections=NHSHH HU           wrapscan
nolist          shell=/bin/sh               wrapmargin=0
magic           shiftwidth=8                nowriteany
nonumber        noshowmatch
```

This chapter discusses only the most commonly used options. For information about the options not covered in this chapter, see **vi**(C) in the *XENIX Reference*.

Setting an Option

To set an option, use the **set** command. For example, to set the *ignorecase* option so that case is *not* ignored in searches, enter:

```
set noignorecase
```

Displaying Tabs and End-of-Line: list

The **list** option causes the "hidden" characters and end-of-line to be displayed. The default setting is **nolist**. To display these characters, enter:

```
:set list
```

Your screen is redrawn. The dollar sign ($) represents end-of-line and ⟨Ctrl⟩i (^I) represents the tab character.

Ignoring Case in Search Commands: ignorecase

By default, case is significant in search commands. To disregard case in searches, enter:

```
:set ignorecase
```

To change this option, enter:

```
:set noignorecase
```

Displaying Line Numbers: number

It is often useful to know the line numbers of a file. To display these numbers, enter:

```
:set number
```

This redraws your screen. Numbers appear to the left of the text.

Printing the Number of Lines Changed: report

The **report** option tells you the number of lines modified by a line-oriented command. For example,

:set report=1

reports the number of lines modified, if more than one line is changed. The default setting is:

report=5

which reports the number of lines changed when more than five lines are modified.

Changing the Terminal Type:term

If you are logged in on a terminal that is a different type than the one you normally use, you can check the terminal type setting by entering:

:set term

Press ⟨Return⟩. See the section "Setting the Terminal Type" for more information about TERM variables.

Shortening Error Messages: terse

After you become experienced with **vi**, you may want to shorten your error messages. To change from the default **noterse**, enter:

:set terse

As an example of the effect of **terse**, when **terse** is set the message:

```
No write since last change, quit! overrides
```

becomes:

```
No write
```

Turning Off Warnings: warn

After you become experienced with **vi**, you may want to turn off the error message that appears if you have not written out your file before a Shell Escape (**:!**) command. To turn these messages off, enter:

```
:set nowarn
```

Permitting Special Characters in Searches: nomagic

The **nomagic** option allows the inclusion of the special characters (. \ $ [ ]) in search patterns without a preceding backslash. This option does *not* affect caret (^) or star (*); they must be preceded by a backslash in searches regardless of **magic**. To set **nomagic**, enter:

```
:set nomagic
```

Limiting Searches: wrapscan

By default, searches in **vi** "wrap" around the file until they return to the place they started. To save time you may want to disable this feature. Use the following command:

```
:set nowrapscan
```

When this option is set, forward searches go only to the end of the file, and backward searches stop at the beginning.

Turning on Messages: mesg

If someone sends you a message with the **write** command while you are in **vi** the text of the message will appear on your screen. To remove the message from your display you must press ⟨Ctrl⟩l. When you invoke **vi**, write permission to your screen is automatically turned off, preventing **write** messages from appearing. If you wish to receive **write** messages while in **vi**, reset this option as follows:

```
:set mesg
```

Mapping Keys

The **map** command maps any character or escape sequence to a command sequence. For example, with the following command defined, when you enter the pound sign (#) in Command mode, **vi** adds a semicolon to the end of the current line.

```
map # A;^[
```

⟨Ctrl⟩[represents the ESC key you must enter to exit from Insert mode. When you create a mapping, use ⟨Ctrl⟩v to escape control characters.

Here is a more complex example:

```
map ^P :w^M:!spell %^M
```

⟨Ctrl⟩p key is mapped to two commands; it writes the file, then executes a shell escape to run the spell checker on the current file (represented by the percent sign). The ⟨Ctrl⟩m represents the ⟨Return⟩ you must enter to execute each command.

Be careful not to map keys that are already defined within **vi**, such as ⟨Ctrl⟩r, which is defined by default to redraw the screen.

You can remove a mapping with the **unmap** command.

Abbreviating Strings

The **abbr** command allows you to avoid typing a frequently used word or phrase by mapping a short string to a longer string. For example, with the following command defined, when you enter "Usa" in Insert mode, **vi** expands the string to "United States of America".

```
:abbr Usa United States of America
```

When you create an abbreviation, it helps to use mixed case (as in "Usa") so that you can still enter "USA" if you need to without it expanding.

You can remove an abbreviation with the **unabbreviate** command.

Customizing Your Environment: The .exrc File

Each time **vi** is invoked, it reads commands from the file named *.exrc* in your home directory. This file sets your preferred options so that they do not need to be set each time you invoke **vi**. A sample *.exrc* file follows:

```
set number
set ignorecase
set nowarn
set report=1
map ^W !}fmt^M
abbr xenix \s-1XENIX\s+1
```

Each time you invoke **vi** with the above settings, your file is displayed with line numbers, case is ignored in searches, warnings before shell escape commands are turned off, and any command that modifies more than one line will display a message indicating how many lines were changed. In addition, the ⟨Ctrl⟩w key is defined to escape to the shell to run a formatting command on the current paragraph, and the string "xenix" is defined to expand to a string containing **troff**(CT) commands that print small capital letters.

Summary of Commands

The following tables contain all the basic commands discussed in this chapter.

Entering vi

Typing this:	Does this:
vi *file*	Starts at line 1
vi +n *file*	Starts at line *n*
vi + *file*	Starts at last line
vi +/pattern *file*	Starts at *pattern*
vi -r *file*	Recovers *file* after a system crash

Cursor Movement

Pressing this key:	Does this:
h	Moves 1 space left
l	Moves 1 space right
⟨Space⟩	Moves 1 space right
w	Moves 1 word right
b	Moves 1 word left
k	Moves 1 line up
j	Moves 1 line down
⟨Return⟩	Moves 1 line down
)	Moves to end of sentence
(	Moves to beginning of sentence
}	Moves to beginning of paragraph
{	Moves to end of paragraph
⟨Ctrl⟩w	Moves to first character of insertion
⟨Ctrl⟩u	Scrolls up 1/2 screen
⟨Ctrl⟩d	Scrolls down 1/2 screen
⟨Ctrl⟩f	Scrolls down one screen
⟨Ctrl⟩b	Scrolls up one screen

Inserting Text

Pressing	Starts insertion:
i	Before the cursor
I	Before first character on the line
a	After the cursor
A	After last character on the line
o	On next line down
O	On the line above
r	On current character, replaces one character only
R	On current character, replaces until ⟨Esc⟩

Delete Commands

Command	Function
dw	Deletes a word
d0	Deletes to beginning of line
d$	Deletes to end of line
3dw	Deletes 3 words
dd	Deletes the current line
5dd	Deletes 5 lines
x	Deletes a character

Change Commands

Command	Function
cw	Changes 1 word
3cw	Changes 3 words
cc	Changes current line
5cc	Changes 5 lines

Search Commands

Command	Function	Example
/and	Finds the next occurrence of *and*	and, stand, grand
?and	Finds the previous occurrence of *and*	and, stand, grand
/^The	Finds next line that starts with *The*	The, Then, There
/[bB]ox/	Finds the next occurrence of *box* or *Box*	
n	Repeats the most recent search, in the same direction	

Search and Replace Commands

Command	Result	Example
:s/pear/peach/g	All *pear*s become *peach* on the current line	
:1,$s/file/directory	Replaces *file* with *directory* from line 1 to the end.	*filename* becomes *directoryname*
:g/one/s//1/g	Replaces every occurrence of *one* with 1.	one becomes 1, oneself becomes 1self, someone becomes some1

Pattern Matching: Special Characters

This character:	Matches:
^	Beginning of a line
$	End of a line
.	Any single character
[]	A range of characters

Leaving vi

Command	Result
:w	Writes out the file
:x	Writes out the file, quits **vi**
:q!	Quits **vi** without saving changes
:!command	Executes *command*
:!sh	Forks a new shell
!!command	Executes *command* and places output on current line
:e *file*	Edits *file* (save current file with **:w** first)

Options

This option:	Does this:
all	Lists all options
term	Sets terminal type
ignorecase	Ignores case in searches
list	Displays tab and end-of-line characters
number	Displays line numbers
report	Prints number of lines changed by a line-oriented command
terse	Shortens error messages
warn	Turns off "no write" warning before escape
nomagic	Allows inclusion of special characters in search patterns without a preceding backslash
nowrapscan	Prevents searches from wrapping around the end or beginning of a file.
mesg	Permits display of messages sent to your terminal with the **write** command

Chapter 3

mail

Introduction

The XENIX **mail** system is a versatile communication facility that allows XENIX users to compose, send, receive, forward, and reply to mail. Users can also create distribution groups and send copies of messages to multiple users. These functions are integrated into XENIX so that all users can quickly and easily communicate with each other.

This chapter is organized to satisfy the needs of both the beginning and advanced user. The first sections discuss basic concepts, tasks, and commands. Later sections discuss advanced topics and provide quick reference to the **mail** program's many functions. The major sections in this chapter are:

Demonstration	Shows new users how to get started.
Basic Concepts	Discusses the fundamental ideas and terminology used in **mail**.
Using mail	Shows how to perform common mailing procedures such as composing, sending, forwarding, and replying to mail.
Commands	Discusses each **mail** command.
Leaving Compose Mode Temporarily	Discusses and gives examples of each command available when composing a message. These commands are called "compose escapes."
Setting Up Your Environment	Discusses the user's mail *startup* file and options that may be set to customize functions.
Using Advanced Features	Discusses advanced features such as using mail as a reminder service and handling a large volume of mail.
Quick Reference	Summarizes all commands, compose escapes, and options.

Demonstration

The **mail** command lets you perform two distinct functions: sending mail and disposing of mail. In this demonstration, we will show you how to send mail to yourself, read a message, delete it, and exit the **mail** program.

Composing and Sending a Message

To begin, enter:

mail self

where *self* is your user name. Next, enter the following lines. Press **RETURN** at the end of each line.

This is a message sent to myself.
I compose a message by entering lines of text.
Press Ctrl-d on a newline to end the message.

As you enter the message you can use "compose escapes" to perform special functions. To get a list of the available compose escapes, enter:

~?

on a new line. To specify a subject, use the ~s escape. For example, enter:

~s Sample subject

To specify a list of people to receive carbon copies use the ~c escape. For example, enter:

~c abel

To view the message as it will appear when you send it, enter:

~p

This will display the following:

```
Message contains:
To: self
Subject: Sample subject
Cc: abel

This is a message sent to myself.
I compose a message by entering lines of text.
Press Ctrl-d on a newline to end the message.
```


Finally, press **Ctrl-d** by itself on a line, to end the message and send it to those that you have mentioned in the *To:* and the *Cc:* fields. You will exit from the **mail** program and return to the XENIX shell. Once you have sent mail, there is no way to undo the act, so be careful.

Reading mail

Your message should have arrived in your system mailbox. To read it, enter:

mail

mail then displays a sign-on message and a list of message headers that look something like this:

```
Mail version 3.0 August 30, 1985.  Type ? for help.
1 message:
  1 self     Fri Aug 31 12:26  7/188 ''Sample subject''
_
```

When there is more than one message in your mailbox, the *most recent* message is displayed at the top of the list. The message at the top of the list has the highest number. The messages are numbered in ascending order from least recent to most recent. The message header includes who sent the message, when it was sent, the number of lines and characters,

and the subject of the message. The underscore prompt prompts you to enter a **mail** command. Now enter:

```
?
```

to get help on all the available **mail** commands. Next, enter:

```
p
```

to see the message that you sent to yourself. **mail** displays the following:

```
From self Fri Aug 20 12:26:52 1985
To: self
Subject: Sample subject

This is a message sent to myself.
I compose a message by entering lines of text.
Press Ctrl-d on a newline to end the message.
```

The message you sent to yourself now contains information about the sender of the message—a line telling who sent the message and when it was sent. The next line tells who the message was sent to. A subject and carbon copy (Cc:) field can be specified by the sender. If they are present, they too are displayed when you read the message.

Note that you can configure your environment so that you are notified whenever new mail is sent to you. To do so, you would have to set the **MAIL** shell variable if you are using the Bourne shell or the **mail** shell variable if you are using the C-shell. For more information, see "The Shell" chapter of the *XENIX User's Guide* and **csh**(C) in the *XENIX Reference.*

Leaving mail

If this message has no real use, you can delete it by entering:

```
d
```

To get out of **mail**, enter:

```
q
```

mail then displays the message

```
0 messages held in /usr/spool/mail/self
```

and returns you to the XENIX shell.

This ends the demonstration. For more detailed information, see the discussions in the following sections.

Basic Concepts

It is much easier to use **mail** if you understand the basic concepts that underlie it. The concepts discussed in this section are:

- Mailboxes
- Messages
- Modes
- Command syntax

Mailboxes

It is useful to think of the **mail** system as modeled after a typical postal system. What is normally called a post office is called the "system mailbox" in this chapter. The system mailbox contains a file for each user in the directory */usr/spool/mail*. Your own personal or "user mailbox" is the file named *mbox* in your home directory. mail sent to you is put in your system mailbox; you may choose to save mail in your user mailbox after you have read it. Note that the user mailbox differs from a real mailbox in several respects:

1. *You* decide whether mail is to be placed in the user mailbox; it is not automatically placed there.

2. The user mailbox is *not* the place where mail is initially routed—that place is the system mailbox in the directory */usr/spool/mail*.

3. mail is not picked up *from* your user mailbox.

Messages

In **mail**, the message is the basic unit of exchange between users. Messages consist of two parts: a heading and a body. The heading contains the following fields:

To: This field is mandatory. It contains one or more valid user names to which you may send mail.

Subject: This optional field contains text describing the message.

Cc: The carbon copy field contains one or more valid names of those who are to receive copies of a message. Message recipients see these names in the received message. This field can be empty.

Bcc: The blind carbon copy field contains the one or more valid names of people who are to receive copies of a message. Recipients do *not* see these names in the received messages. This field can be empty.

Return-receipt-to:
The return receipt to: field contains the valid name or names of those who are to receive an automatic acknowledgement of the message. This field can be empty.

The body of a message is text exclusive of the heading. The body can be empty.

Modes

Often, the biggest hurdle to using **mail** is understanding what modes of operation are available. This section discusses each mode.

When you invoke **mail** you are using the shell. If you want to mail a letter without entering **mail** command mode, you can do so by entering:

mail john < letter

Here, the file *letter* is sent to the user *john*.

Note

Be very careful when mailing a file with the input redirection symbol (<). If you accidentally enter the output redirection symbol (>), you will overwrite the file, destroying its contents.

You can enter a message from your shell by entering:

mail john

Next, enter the text of your message as follows:

This is the text of the message.

Press RETURN to start a new line, then Ctrl-d to send the message.

Messages such as the one above are created in *compose mode*. When entering text in compose mode, there are several special keys associated with line editing functions: these are the same special characters that are available to you when executing normal XENIX commands. For example, you can kill the line you are editing by entering **Ctrl-u**, normally the kill character. To backspace, press the **BACKSPACE** Key or **Ctrl-h**.

From compose mode, you can issue commands called compose escapes. These are also called *tilde escapes* because the command letters are preceded by a tilde (~). When you execute these commands you are temporarily leaving or escaping from compose mode; hence the name. Note that once you have pressed **RETURN** to end a line, you cannot change that line from within compose mode. You must enter edit mode in order to change that line.

The most common way of using **mail** is just to enter:

mail

If you have mail waiting, this command will automatically place you in *command mode*. In this mode, you are prompted by an underscore for commands that permit you to manage your mail. If you have no mail waiting, you see the message *no messages* and are returned to the XENIX prompt.

You can enter *edit mode* from either compose mode or command mode. In edit mode, you edit the body of a message using the full capabilities of an editor. To enter edit mode from command mode, use either the **e** or **edit** command to enter **ed**, or the **v** or **visual** command to enter **vi**. (**Vi** may not be available on your system.) To enter edit mode from compose mode, use the compose escapes ~e and ~v, respectively.

Message-Lists

Many **mail** commands take a list of messages as an argument. A *message-list* is a list of message numbers, ranges, and names, separated by spaces or tabs. Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters ^, ., or $, *which specify the first, current, or last undeleted* message, respectively. Here, relevant means *not deleted.*

A range of messages is two message numbers separated by a dash. To display the first four messages on the screen, enter:

```
p 1-4
```

To display all the messages from the current message to the last message, enter:

```
p .-$
```

A *name* is a user name. Messages can be displayed by specifying the name of the sender. For example, to display each message sent to you by *john,* enter:

```
p john
```

As a shorthand notation, you can specify star (*) to get all *undeleted* messages. For example, to display all messages except those that have been deleted, enter:

```
p*
```

To delete all messages, enter:

```
d*
```

To restore all messages, enter:

```
u*
```

All three of these commands are described later in detail in the section "Commands."

Headers

When you enter **mail**, a list of message *headers* is displayed. A header is a single line of text containing descriptive information about a message. (Note that we use the word *heading* to describe the first part of a message, and *header* to describe **mail**'s one-line description of a message.) The information includes:

- The number of the message
- The sender
- The date sent
- The number of characters and lines
- The subject (if the message contains a *Subject:* field)

Message headers are displayed in *windows* with the **headers** command. A header window contains no more than 18 headers. If there are fewer than 18 messages in the mailbox, all are displayed in one header window. If there are *more* than 18 messages, then the list is divided into an appropriate number of windows. You can move forward one window at a time with the command:

```
headers +
```

and move backward one window at a time with the command:

```
headers -
```

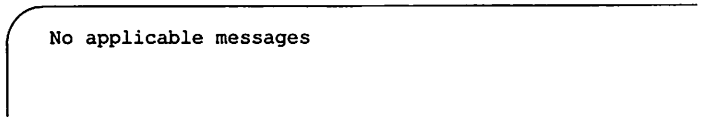
commands.

Command Syntax

Each **mail** command has its own syntax. Some take no arguments, some take only one, and others take several arguments. The more flexible commands, such as **print**, accept combinations of message-lists and user names. For these commands, **mail** first gathers all message numbers and ranges, then finds all messages from any specified user names. The full message-list is the intersection of these two sets of messages. Thus, the message-list "4-15 miller" matches all messages between 4 and 15 that are from miller.

Each **mail** command is entered on a line by itself, and any arguments follow the command word. The command need not be entered in its entirety—the first command that matches the entered prefix is used. For example, you can enter "p" instead of "print" for the **print** command and "h" instead of "headers" for the **headers** command.

After the command itself is entered, one or more spaces should be entered to separate the command from its arguments. If a **mail** command does not take arguments, any arguments you give are ignored and no error occurs. For commands that take message-lists as arguments, if no message-list is given, the last message printed is used. If it does not satisfy the requirements of the command, the search proceeds forward. If there are no messages ahead of the current message, the search proceeds backwards, and if there are no valid messages at all, **mail** displays:



```
No applicable messages
```

Using mail

This section describes how to perform some basic tasks when using **mail**. More detailed discussions of each of these commands are presented in later sections.

Entering and Exiting mail

To begin a session with **mail**, enter:

mail

The headers for each received message are then displayed one screenful at a time. To display the next screenful of headers (if any), enter:

h+

To end the **mail** session, use the **quit** (q) command. All messages remain in the system mailbox unless they have been deleted with the **delete** (d) command, saved with the **save** or **write** command, or held in your user mailbox with the **mbox** command. Deleted messages are discarded. The **-f** command line option causes **mail** to read in the contents of *mbox*. Optionally, a filename may be given as an argument to **-f**, so that the specified file is read instead. When you **quit**, **mail** writes all messages back to this file.

If you send mail over a noisy phone line, you will notice that many of the bad characters turn out to be RUBOUT or DEL character. These characters cause **mail** to abort messages. To deal with this annoyance, you can invoke **mail** with the **-i** option which causes these bad characters to be ignored.

Sending mail

To send a message, invoke **mail** with the names of the people and groups you want to receive the message. Next, enter your message. When you are finished, press Ctrl-d at the beginning of a line. The message is automatically sent to the specified people. While entering the text of your message, you can escape to an editor or perform other useful functions with compose escapes. The section "Composing mail," describes some features of **mail** available to help you when composing messages.

If you have a file that contains a written message, you can send it to sam, bob, and john by entering:

mail sam bob john < letter

where *letter* is the name of the file you are sending.

Note

Be very careful when mailing a file with the input redirection symbol (<). If you accidentally enter the output redirection symbol (>), you will overwrite the file, destroying its contents.

If **mail** cannot be delivered to a specified address, you will either be notified immediately, in which case a copy of the undeliverable message is appended to the file *dead.letter*, or you will be notified via return mail, in which case a copy is included in the return mail message.

Sending Mail to Remote Sites

You can send mail to users on remote computer sites that are networked to your own site. The network can either be a Micnet network or a UUCP network. Ask your system administrator if you are not sure which network the site you want to mail to uses.

If the site you want to send mail to is a Micnet site, you would enter the following command to mail to a user on that site:

mail *site-name:user*

Note that the site name is followed by a colon (:).

For example, to send mail to stevem on the Micnet computer named *obie*, you would enter the following command:

mail obie:stevem

After entering this command, you would continue with **mail** just as if you were sending mail to a local user.

You can also send mail to users on remote UUCP sites. To find out which UUCP sites your computer communicates with, enter the following command at the XENIX prompt:

uuname

A list of site names is displayed.

To send mail to a user on a UUCP site, enter the following command:

mail *site-name*!*user*

The site name must be followed by an exclamation mark (!). You can have several site names on a command line. Be sure to follow each one with an exclamation mark.

For example, to send mail to user markt on site *bowie*, you would enter the following command:

mail bowie!markt

You would then proceed to use **mail** just as if you were mailing a local user.

As another example, suppose your site talked to UUCP site *bowie* and that *bowie* talked to UUCP site *bradley*. You could send mail to user cindy on *bradley* by entering the following command:

mail bowie!bradley!cindy

Note

If you are using the C-shell, you must "escape" exclamation marks with the backslash (\). A C-shell user would enter the above command as follows:

mail bowie\!bradley\!cindy

For more information on communicating with remote sites, see the "Communicating with Other Sites" chapter in this guide.

Reading mail

To read messages sent to you, enter:

mail

mail then checks your mail out of the system mailbox and prints out a one-line header of each message, one screenful at a time. Enter "h+" to view the next screenful. The most recent message is initially the first message (numbered highest, because messages are numbered chronologically) and may be printed using the **print** command. You can move forward one message by pressing RETURN or entering "+". To move forward *n* messages use "+*n*". You can move backwards one message with the "-" command or move backwards *n* messages and print with "-*n*". You can also move to any arbitrary message and print it by entering its number.

If new messages arrive while you are in **mail**, the following message appears:

```
New mail has arrived--type 'restart' to read.
```

Enter:

restart

and the headers of the new messages are displayed.

Disposing of mail

After examining a message you can delete it with the **delete** (d) command, reply to it with the **reply** (r) command, forward it with the **forward** (f) command, or skip to the next message by pressing RETURN. Deletion causes the mail program to forget about the message. This is not irreversible; the message can be *undeleted* with the **undelete**(u) command by entering:

u *number*

Composing mail

To compose mail, you must enter compose mode. Do this from XENIX command level by entering:

mail john

where john is the name of a user to whom you want to send mail. From **mail** command mode, you can enter compose mode with the **mail, reply,** or **Reply** commands. Once in compose mode, the text that you enter is appended one line at a time to the body of the message you are sending. Normal line editing functions are available when entering text, including Ctrl-u to kill a line and Backspace to back up one character. Note that when you enter two interrupts in a row (i.e., pressing INTERRUPT twice), your composition is aborted.

While you are composing a message, **mail** treats lines beginning with the tilde character (~) in a special way. This character introduces commands called compose escapes. For example, entering:

~m

by itself on a line places a copy of the most recently printed message inside the message you are composing. The copy is shifted right one tabstop.

Other escapes set up heading fields, add and delete recipients to the message, allow you to escape to an editor, let you revise the message body, or run XENIX commands. To get a list of the available compose escapes when in compose mode, enter:

~?

See also "Leaving Compose Mode Temporarily," later in this chapter.

Forwarding mail

To forward a message, use the **forward** (f) command. For example, enter:

f john

to forward the current message to *john*. John will receive a copy of the current message, along with a new header indicating that it came from you. The copy is shifted right one tabstop.

The **Forward** (F) command works just like its lowercase counterpart, except that the forwarded message is not shifted right one tabstop.

Replying to mail

You can use the **reply** command to set up a response to a message, automatically addressing a reply to the person who sent the original message. You can enter text and send the message by pressing Ctrl-d on a line by itself. The **Reply** command works just like its lowercase counterpart, except that the message is sent to others named in the original message's *To:* and *Cc:* fields.

Specifying Messages

Commands such as **print** and **delete** can be given a message-list argument to apply to several messages at once. Thus "delete 2 3" deletes messages 2 and 3, while "delete 1-5" deletes messages 1 through 5. A star (*) addresses all messages, and a dollar sign ($) addresses the last (highest numbered) message. The **top** (t) command displays the first five lines of a message; hence, you can enter:

```
top *
```

to display the first five lines of every message. Message-lists can contain combinations of lists, ranges, and names. For example, the following command displays all messages from tom or bob and numbered 2, 4, 10, 11, or 12:

```
p tom bob 2 4 10-12
```

Creating Mailing Lists

You can create personal mailing lists so that, for example, you can send mail to *cohorts* and have it go to a group of people. Such lists are defined by placing an *alias* line like:

```
alias cohorts bill bob barry
```

in the file *.mailrc* in your home directory. The current list of such aliases can be displayed with the **alias (a) mail** command. Personal aliases are expanded in mail sent to others so that they will be able to **Reply** to each individual recipient. For example, the *To:* field in a message sent to *cohorts* will read:

```
To: bill bob barry
```

and not:

```
To: cohorts
```

Normally, system-wide aliases are available to all users. These are installed by whoever is in charge of your system. For more information, see the section "Using Advanced Features," later in this chapter.

Setting Options

mail has several options that you can **set** from **mail** command mode or in the file *.mailrc* in your home directory. For example, "set askcc" enables the **askcc** switch and causes prompting for additions to the *Cc:* field when you finish composing a message. These and other options are discussed in the section "Setting Up Your Environment: The .mailrc File."

Commands

This section describes each of the commands available to you in **mail** command mode. The examples in this section assume you have invoked **mail** and that you have several messages you want to dispose of. Note that in general, **mail** commands can be invoked with either the name of the command or a one- or two-character mnemonic abbreviation. In the text of the command descriptions below, this mnemonic abbreviation is enclosed in parentheses after the name of the command. All commands are printed in boldface, except in the examples.

Getting Help: help and ?

The **help** (?) command displays a brief summary of all **mail** commands, so if you ever get stuck when you are in **mail** command mode, enter:

```
?
```

or:

```
help
```

Reading mail: p, +, -, and *restart*

To look at a specific message, use the **print** (p) command. For example, pretend you have a header-list that looks like this:

```
3 john Wed Sep 21 09:21 26/782 ''Notice''
2 sam Tue Sep 20 22:55 6/83 ''Meeting''
1 tom Mon Sep 19 01:23 6/84 ''Invite''
```

Reading from the left, each header contains the message number, who sent it, the day, date, and time it was sent, the number of lines and characters in the message, and its subject.

To examine the second message, enter:

```
p 2
```

This might cause **mail** to respond with:

```
Message  2:
From sam  Tue June 20 22:55 1985
Subject: Meeting

Meeting everyone, please do not forget!
```

To look at message 3, enter:

```
-
```

or to look at message 1, enter:

```
+
```

The commands + and - execute relative to the last message referred to, which in our example was 2. For large numbers of messages, you can skip forward and backward by the number of messages specified as an argument to + and -. For example, entering:

```
+3
```

skips forward three messages. If you enter:

```
p *
```

then all messages are displayed, since the star (*) matches all messages.

Pressing RETURN displays the next message in the header-list. You can always go to a message and print it by giving its message number or one of the special characters, caret (^), dot (.), or dollar sign ($). In the example where message 2 is the current message, to display the current message, enter:

```
.
```

To display message 1, enter:

```
^
```

To display message 3, enter:

```
$
```

When new mail arrives while you are in **mail**, the message "New mail has arrived—type 'restart' to read." If you wish to read the new messages, enter:

```
restart
```

The headers of the new messages appear.

Finding Out the Number of the Current Message: =

The **number** (=) command displays the message number of the current message. It takes no arguments.

Displaying the First Five Lines : t

The **top** (t) command takes a message-list and displays the first five lines of each addressed message. For example:

```
top 2-12
```

displays the first five lines of each of the messages 2 through 12. Note that the number of lines displayed by **top** can be set with the **toplines** option.

Displaying Headers: h

The **headers** (h) command displays header windows or lists of headers. A header window contains no more than 18 headers. With no argument, the **headers** command displays a header window in which the current message header is displayed at the center of the window.

To examine the next set of 18 headers, enter:

```
h +
```

To examine the previous set, enter:

```
h -
```

Both plus and minus take an optional numeric argument that indicates the number of header windows to move forward or backward before printing. If a message-list is given, then the **headers** command displays the header line for each message in the list, disregarding all windowing. For example:

```
h joe
```

displays all the message headers from joe. The following are some characteristics of the header-list:

- Deleted messages do not appear in the listing.
- Messages saved with the **save** command are flagged with a star (*).
- Messages to be saved in your user mailbox are flagged with an "M".
- If the *autombox* option is set, messages held with the **hold** command are flagged with an "H".

Deleting Messages: d and dp

Unless you indicate otherwise, each message you receive is automatically saved in the system mailbox when you quit **mail**. Often, however, you do not want to save messages you have received. To delete messages, use the **delete** (d) command. For example:

```
delete 1
```

prevents **mail** from retaining message 1 in the system mailbox. The message will disappear altogether, along with its number.

The **dp** command deletes the current message and displays the next message. It is useful for quickly reading and disposing of mail. Using **dp** is the same as using the **d** command with the *autoprint* option set. See also the **undelete** command, below.

Undeleting Messages: u

The **undelete** (u) command causes a message that has been previously deleted with **d** or **dp** to reappear as if it had never been deleted. For example, to undelete message 3, enter:

```
u3
```

You cannot undelete messages from previous **mail** sessions; they are permanently deleted.

Leaving mail : q and x

When you have read all your messages, you can leave **mail** with the **quit** (q) command. All messages are held in your system mailbox, except the following:

- Deleted messages, which are discarded irretrievably.
- Messages marked with the **mbox** command, which are saved in *mbox* in your home directory (that is, your user mailbox).
- Messages saved with the **save** and **write** commands are deleted from the system mailbox. Forwarded messages are *not* deleted.

Note that if the *autombox* option is set, messages that you have read are automatically saved in your user mailbox. If you wish to leave **mail** quickly without altering either your system or user mailbox, you can use the **exit** (x) command. This returns you to the shell without changing anything: no messages are deleted or saved. Files that you invoke with the **mail -f** switch are unaffected as well.

Saving Your mail: s

The **save** (s) command lets you save messages to files other than *mbox*. By using **save**, you can organize your mail by putting messages in appropriate files. The **save** command writes out each message to the file given as the last argument on the command line. For example, the following command appends messages 1-5 to the file *letters :*

```
s 1-5 letters
```

The file *letters* is created if it does not already exist. Saved messages are not automatically retained in the system mailbox when you quit, nor are

they selected by the **print** command described above, unless explicitly requested. Each saved message is marked with a star (*).

Save writes out the entire message, including the *To:*, *Subject:*, and *Cc:* fields. In comparison, the **write** command, discussed below, writes out only the bodies of the specified messages.

Saving Your mail: w

The **write** (w) command writes out *the body* of each message to the file given as the last argument on the command line. Each written message is marked with a star (*). The syntax is similar to that of the **save** command. For example,

```
w 3-17 john elliot book
```

writes out the bodies of all messages from john and elliot in the number range 3-17. They are concatenated to the end of the file named *book*.

Saving Your mail: mb

The **mbox** (mb) command marks each message specified in a message-list, so that all are saved in the user mailbox when a **quit** command is executed. Message headers are marked with an "M" to show that they are to be saved in *mbox*.

Saving Your mail: ho

The **hold** (ho) command takes a message-list and marks each message so that it is saved in your system mailbox instead of deleted or saved in *mbox* when you quit. Saving of files in the system mailbox happens by default, so use **hold** *only* when you have also set the **autombox** option.

Printing Your mail on the Lineprinter: l

The **lpr** (l) command paginates and prints out messages to the lineprinter. It takes a message-list as its argument, then paginates and prints out each message. For example:

```
l doug
```

prints out each message from the user doug on the lineprinter.

Sending mail: m

To send mail to a user, use the **mail** (m) command. This sends mail in the manner described for the **reply** command, except that you supply a list of recipients either as an argument or by entering them in the *To:* field. All compose escapes work in **mail**. Note that the **mail** command is in most ways identical to entering *mail users* at the XENIX command level.

Replying to mail: r and R

Often, you want to deal with a message by responding to its author right away. The **reply** (r) command is useful for this purpose: it takes a message-list and sends mail to the author of each message. The original message's subject field is copied as the reply's subject. Each message is created in compose mode; thus all compose escapes work in **reply**, and messages are terminated by pressing Ctrl-d.

The **Reply** (R) command works just like its lowercase counterpart, except that copies of the reply are also sent to everyone shown in the original message's *To:* and *Cc:* fields.

Forwarding mail: f and F

To forward a copy of a message, use the **forward** (f) command. This causes a copy of the current message to be sent to the specified users. The message is marked as saved, and then deleted from the system mailbox when you exit **mail**. For example, to forward the current message to someone whose login name is john, enter:

```
f john
```

John will receive the forwarded message, along with a heading showing that you are the one who forwarded it. The forwarded message is indented one tab stop inside the new message. An optional message number can also be given. For example:

```
f 2 john bill
```

forwards message 2 to john and bill.

The **Forward** (F) command is identical to the lowercase **forward** command, except that the forwarded message is not indented.

Creating mailing Lists: a

The **alias** (a) command links a group of names with the single name given by the first argument, thus creating a mailing list. For example, you could enter:

```
alias beatles john paul george ringo
```

so that whenever you used the name *beatles* in a destination address (as in "mail beatles"), it would be expanded so that you are really referring to the four names aliased to *beatles*. With no arguments, **alias** displays all currently-defined aliases. With one argument, it prints out the users defined by the given alias.

You will probably want to define aliases in the startup file, *.mailrc*, so that you do not have to redefine them each time you invoke **mail**. See the section "Setting Up Your Environment: The .mailrc File," for more information.

Setting and Unsetting Options: se and uns

mail switch and string options can be set with the **mail** commands **set** and **unset**. A switch option is either on or off (set or unset). String options are strings of characters that are assigned values with the syntax *option=string*. Multiple options may be specified on a line. It is most useful to place set and unset commands in the file *.mailrc* in your home directory, where they become your own personal default options when you invoke **mail**. For example, you might have a **set** command that looked like this:

```
set dot metoo toplines=10 SHELL=/usr/bin/sh
```

The options *dot* and *metoo* are switch options; *toplines* and *SHELL* are string options.

The command

```
set ?
```

displays a list of the available options. See the section "Setting Up Your Environment," for descriptions of these options.

Editing a Message: e and v

Invoke the **edit** command to edit individual messages while using the text editor. The **edit** command takes a message list and processes each message in turn by writing it to a temporary file. The editor, *ed,* is then automatically invoked so that you can edit the temporary file. When you finish editing the message, write the message out, then quit the editor. **mail** reads the message back into the message buffer and removes the temporary file.

It is often useful to be able to invoke either a line or visual editor, depending on the type of terminal you are using. To invoke **vi,** you can use the **visual** (v) command. The operation of the **visual** command is otherwise identical to that of the **edit** command.

Executing Shell Commands: sh and !

To execute a shell command without leaving **mail,** precede the command with an exclamation point. For example:

```
!date
```

displays the current date without leaving **mail.** To enter a new shell, enter:

```
sh
```

To exit from this new shell and return to **mail** command mode, press Ctrl-d.

Finding the Number of Characters in a Message: si

The **size** (si) command displays the number of characters in each message in a message-list. For example, the command: "si 1-4" might display:

```
4:  234
3:  1000
2:  23
1:  456
```

Changing the Working Directory: cd

The **cd** command changes the working directory to the name of the directory you give it as an argument. If no argument is given, the directory is changed to your home directory. This command works just like the normal XENIX **cd** command. (Note that exiting **mail** returns you to the directory from which you entered **mail**; thus the **mail cd** command works only within **mail**.) You may want to place a **cd** command in your *.mailrc* file so that you always begin executing **mail** from within the same directory.

Reading Commands From a File: so

The **source** (so) command reads in **mail** commands from named file. Normally, these commands are **alias**, **set**, and **unset** commands.

Leaving Compose Mode Temporarily

While composing a message to be sent to others, it is often useful to print a message, invoke the text editor on a partial message, execute a shell command, or perform some other function. **mail** provides these capabilities through *compose escapes* (sometimes called *tilde escapes*) which consist of a tilde (~) at the beginning of a line, followed by a single character that specifies the function to be performed. These escapes are available *only* when you are composing a new message. They have no meaning when you are in **mail** command mode. The available compose escapes are described below.

Getting Help: ~?

The help escape is the first compose escape you should know because it tells you about all the others. For example, if you enter:

```
~?
```

a brief summary of the available compose escapes is displayed on your screen. Note that ~h prompts for heading fields and and does *not* give help.

Printing the Message: ~p

To print the current text of a message you are composing, enter:

```
~p
```

This prints a line of dashes and the heading and body of the message so far.

Editing the Message: ~e and ~v

If you are dissatisfied with a message as it stands, you can edit the message by invoking the editor, **ed**, with the editor escape, ~e. This causes the message to be copied into a temporary file so that you can edit it. Similarly, the ~v escape causes the message to be copied into a temporary

file so that you can edit it with the vi editor. After modifying the message to your satisfaction, write it out and quit the editor. **mail** responds:

```
(continue)
```

after which you may continue composing your message.

3 Editing Headers: ~t, ~c, ~b, ~s, ~R and ~h

To add additional names to the list of message recipients, enter the escape:

~t name1 name2 ...

You can name as many additional recipients as you wish. Note that users originally on the recipient list will still receive the message: you cannot remove anyone from the recipient list with **~t**. To remove a recipient, use the **~h** command, which is discussed later in this section.

You can replace or add a subject field by using the **~s** escape:

~s *line-of-text*

This replaces any previous subject with *line-of-text*. The subject, if given, appears near the top of the message, prefixed with the heading *Subject:*. You can see what the message looks like by using **~p**, which displays all heading fields along with the body of the text.

You may occasionally prefer to list certain people as recipients of carbon copies of a message rather than direct recipients. The escape:

~c name1 name2 ...

adds the named people to the *Cc:* list. The escape:

~cc name1 name2 ...

performs an identical function. Similarly, the escape:

~b name1 name2 ...

adds the named people to the *Bcc:* (Blind carbon copy) list. The people on this list receive a copy of the message, but are not mentioned anywhere in the message you send. Remember that you can always execute a ~p escape to see what the message looks like.

The escape:

~R

adds or changes the person or persons named in the *return-receipt-to:* field.

The recipients of the message are given in the *To:* field; the subject is given in the *Subject:* field, carbon copy recipients are given in the *Cc:* field and the return receipt recipient in the *Return-receipt-to:* field. If you wish to edit these in ways impossible with the ~t, ~s, ~c, and ~R escapes, you can use:

~h

where h stands for "heading." The escape **~h** displays *To:* followed by the current list of recipients and leaves the cursor at the end of the line. If you enter ordinary characters, they are appended to the end of the current list of recipients. You can also use the normal XENIX command line editing characters to edit these fields, so you can erase existing heading text by backspacing over it.

When you press RETURN, **mail** advances to the *Subject:* field, where the same rules apply. Another RETURN brings you to the *Cc:* field, another brings you to the *Bcc:* field, and yet another to the *Return-receipt-to:* field. Each of these fields can be edited in the same way. Finally, another RETURN leaves you appending text to the end of your message body. As always, you can use ~p to print the current text of the heading fields along with the body of the message.

Adding a File to the Message: ~r and ~d

It is often useful to be able to include the contents of some file in your message. The escape:

~r filename

is provided for this purpose, and causes the named file to be appended to your current message. **mail** complains if the file does not exist or cannot be read. If the read is successful, **mail** displays the number of lines and characters appended to your message.

As a special case of ~r, the escape:

~d

reads in the file *dead.letter* in your home directory. This is often useful because **mail** copies the text of your message buffer to *dead.letter* whenever you abort the creation of a message. You can abort the message by entering two consecutive interrupts or by entering a **~q** escape.

Enclosing Another Message: ~m and ~M

If you are sending mail from within mail's command mode, you can insert a message, that was previously sent to you, into the message that you are currently composing. For example, you might enter:

~m 4

This reads message 4 into the message you are composing, shifted right one tab stop. The escape:

~M 4

performs the same function, but with no right shift. You can name any nondeleted message or list of messages.

Saving the Message in a File: ~w

To save the current text of a message body in a file, use:

~w filename

mail writes out the message body to the specified file, then displays the number of lines and characters written to the file. The ~w escape does *not* write the message heading to the file.

Leaving mail Temporarily: ~! and ~|

To temporarily escape to the shell, use the escape:

~!*command*

This executes *command* and returns you to **mail** compose mode without altering your message. If you wish to filter the body of your message through a shell command, use:

~|*command*

This pipes your message through the command and uses the output as the new text of your message. If the command produces no output, **mail** assumes that something is wrong. It retains the old version of your message, and displays:

```
(continue)
```


Escaping to mail Command Mode: ~:

To temporarily escape to **mail** command mode, use either of the escapes:

~:*mail-command*

~_*mail-command*

You can then execute any **mail** command that you want. Note that this escape will not work in most cases if you enter compose mode from the XENIX shell. It depends on the command used (**set** and **unset** will work), but most commands that involve message lists are not allowed. You will receive the message:

```
May not execute cmd while composing
```

Placing a Tilde at the Beginning of a Line: ~~

If you wish to send a message that contains a line beginning with a tilde, you must enter it twice. For example, entering:

~~This line begins with a tilde.

appends:

~This line begins with a tilde.

to your message. The escape character can be changed to a different character with the *escape* option. (For information on how to set options, see the section "Setting Up Your Environment: The .mailrc File.") If the escape character is not a tilde, then this discussion applies to that character and not the tilde.

Setting Up Your Environment: The .mailrc File

Whenever **mail** is invoked, it first reads the file */usr/lib/mail/mailrc* then the file *.mailrc* in the user's home directory. System-wide aliases are defined in */usr/lib/mail/mailrc*. Personal aliases and **set** options are defined in *.mailrc*. The following is a sample *.mailrc* file:


```
# number sign introduces comments

# personal aliases office and cohorts are defined below

alias office bill steve karen
alias cohorts john mary bob beth mike

# set dot lets messages be terminated by period on new line

# set askcc says to prompt for Cc: list after composing message

set dot askcc

# cd changes directory to different current directory

cd
```

The Subject Prompt: asksubject

The **asksubject** switch causes prompting for the subject of each message before you enter compose mode. If you respond to the prompt with a **RETURN**, then no subject field is sent.

The CC Prompt: askcc

The **askcc** switch causes prompting for additional carbon copy recipients when you finish composing a message. Responding with a RETURN signals your satisfaction with the current list. Pressing INTERRUPT displays:

```
interrupt
(continue)
```

so that you can return to editing your message.

Printing the Next Message: autoprint

The **autoprint** switch causes the **delete** command to behave like **dp**. After deleting a message, the next message in the list is automatically printed. Printing also occurs automatically after execution of an **undelete** command.

Listing Messages in Chronological Order

The *chron* switch causes messages to be listed in chronological order. By default, messages are listed with the most recent first. Set *chron* when you want to read a series of messages in the order they were received.

The *mchron* switch, like *chron*, displays messages in chronological order, but lists them in the opposite order, that is, highest-numbered, or most recent, first. This is useful if you keep a large number of messages in your mailbox and you wish to list the headers of the most recently received mail first but read the messages themselves in chronological order.

Using the Period to Send a Message: dot

The *dot* switch lets you use a period (.) as an end-of-transmission character, as well as **Ctrl-d**. This option is available for those who are used to this convention when editing with the editor, **ed**.

Sending mail While in mail: execmail

It is often desirable to reply to a piece of mail, or send mail while reading your mail file. This process is speeded up by the use of the *execmail* option. It causes the underbar prompt to return before **mail** is finished being sent. This frees the user to continue while **mail** performs mailing functions in the background.

Including Yourself in a Group: metoo

Usually, when a group is expanded that contains the name of the sender, the sender is removed from the expansion. Setting the **metoo** option causes the sender to be included in the group.

Saving Aborted Messages: save

The **nosave** switch prevents aborted messages from being appended to the file *dead.letter* in your home directory; messages are saved by default. You can abort messages when you are in compose mode by entering two interrupts or a ~**q** compose escape.

Printing the Version Header: quiet

The **quiet** switch suppresses the printing of the version header when **mail** is first invoked.

Choosing an Editor: The EDITOR String

The *EDITOR* string contains the pathname of the text editor to use in the **edit** command and ~e escape. If not defined, then the default editor is used. For example:

```
set EDITOR=/bin/ed
```

Choosing an Editor: The VISUAL String

The *VISUAL* string contains the pathname of the text editor used in the **visual** command and ~v escape. For example:

```
set VISUAL=/bin/vi
```

By default, **vi** is the editor used.

Choosing a Shell: The SHELL String

The *SHELL* string contains the name of the shell to use in the ! command and the ~! escape. A default shell is used if this option is not defined. For example:

```
set SHELL=/bin/sh
```

Changing the Escape Character: The escape String

The *escape* string defines the character to use in place of the tilde (~) to denote compose escapes. For example:

```
set escape=*
```

With this setting, the asterisk becomes the new compose escape character.

Setting Page Size: The page String

The *page* string causes messages to be displayed in pages of size *n* lines. You are prompted with a question mark between pages. Pressing RETURN causes the next page of the current message to be displayed. By default this paging feature is turned off.

Saving Outgoing mail: The record String

The *record* string sets the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not copied and saved. For example:

```
set record=/usr/john/recordfile
```

With this setting, all outgoing mail is automatically appended to the file */usr/john/recordfile*.

Keeping mail in the System mailbox: autombox

The *autombox* switch determines whether messages remain in the system mailbox when you exit **mail**. If you set *autombox*, the examined messages are automatically placed in the *mbox* file in your home directory (your user mailbox). They are *removed* from the system mailbox when you quit.

Changing the top Value: The toplines String

The *toplines* string sets the number of lines of a message to be displayed with the **top** command. By default, this value is five. For example:

```
set toplines=10
```

With this setting, ten lines of each message are displayed when the **top** command is used.

Sending mail Over Telephone Lines: ignore

The *ignore* switch causes interrupt signals from your terminal to be ignored and echoed as at-signs (@). This switch is normally used only when communicating with **mail** over telephone lines.

Using Advanced Features

This section discusses advanced features of **mail** useful to those with some existing familiarity with the XENIX **mail** system.

Command Line Options

One very useful command line option to **mail** is the -s "subject" switch. You can specify a subject on the command line with this switch. For example, you could send a file named *letter* with the subject line, "Important Meeting at 12:00", by entering the following:

mail -s "Important Meeting at 12:00" john bob mike <letter

To include other header fields in your message, you can use the following options:

-b *user* Adds the blind carbon copy field to the message header.

-c *user* Adds the carbon copy field to the message header.

-r *user* Adds the return-receipt to: field to the message header.

None of the above options may be specified more than once on a mail command line. If multiple arguments are required for an option, the entire argument set must be enclosed in quotes, as in:

mail -r "meeting" -b singleuser -c "x y z" user user2

mail also allows you to edit files of messages by using the **-f** switch on the command line. For example:

mail -f *filename*

causes **mail** to edit *filename* and the command:

mail -f

causes **mail** to read *mbox* in your home directory. All the **mail** commands except **hold** are available to edit the messages. When you enter the **quit** command, **mail** writes the updated file back.

If you send mail over a noisy phone line, you may notice that bad characters are transmitted. These are characters that abort messages: RUBOUT and DEL. You can invoke **mail** with the **-i** switch to ignore these bad characters.

When you enter the mail program (as opposed to sending a message from command level), two command line options are available:

-R Makes the mail session read-only, preventing alteration of the mail being read.

-u *user* Reads in *user's* mail instead of your own.

Using mail as a Reminder Service

Besides sending and receiving mail, you can use **mail** as a reminder service. Several XENIX commands have this idea built in to them. For example, the XENIX **lp** command's **-m** switch causes mail to be sent to the user after files have been printed on the lineprinter. XENIX automatically examines the file named *calendar* in each user's home directory and looks for lines containing either today or tomorrow's date. These lines are sent by **mail** as a reminder of important events.

If you program in the shell command language, you can use **mail** to signal the completion of a job. For example, you might place the following two lines in a shell procedure:

```
biglongjob
echo "biglongjob done" | mail self
```

You can also create a logfile that you want to mail to yourself. For example, you might have a shell procedure that looks like this:

```
dosomething >logfile
mail self <logfile
```

For information about writing shell procedures, see "The Shell" chapter in this Guide.

Handling Large Amounts of mail

Eventually, you will face the problem of dealing with an accumulation of messages in your user mailbox. There are a number of strategies that you can employ to solve this problem concerning space in your mailbox file. Keep in mind the dictum:

> When in doubt, throw it out.

This means that you should only save *important* mail in your user mailbox. If your mailbox file becomes large, you must periodically examine its contents to decide whether messages are still relevant. To save space, consider summarizing very long messages.

The previously mentioned measures are not always helpful enough in organizing the many messages that you are likely to receive. Another effective approach is to save mail in files organized by sender, by topic, or by a combination of the two. Create these files in a separate **mail** directory; you can access these mailbox files with the **mail** -f *filename* switch. However, be forewarned—this approach to organizing mail quickly eats up disk space.

Maintenance and Administration

The following is a list of the programs and files that make up the XENIX **mail** system:

/usr/bin/mail	mail program
/usr/lib/mail/mailrc	mail system initialization file
/usr/spool/mail/*	System mailbox files
/usr/*name*/dead.letter	File where undeliverable mail is deposited
/usr/*name*/mbox	User mailbox
/usr/*name*/.mailrc	User mail initialization file
/usr/lib/mail/mailhelp.cmd	mail command help file
/usr/lib/mail/mailhelp.esc	mail compose escape help file
/usr/lib/mail/mailhelp.set	mail option help file

/usr/lib/mail/aliases	System-wide aliases
/usr/lib/mail/aliases.hash	System-wide alias database
/usr/lib/mail/faliases	Forwarding aliases
/usr/lib/mail/maliases	Machine aliases
/usr/lib/mail/maliases.hash	Optional machine aliases database

A system-wide distribution list is kept in */ usr/ lib/ mail /aliases*. A system administrator is usually in charge of this list. These aliases are kept in a vastly different syntax from *.mailrc*, and are expanded when mail is sent. You will normally need special permission to change system-wide aliases.

Quick Reference

The following sections provide quick reference to the available commands, compose escapes, and options.

Command Summary

Given below are the name and syntax for each command, the abbreviated form (in brackets), and a short description. Many commands have optional arguments; most can be executed without any arguments at all. In particular, commands that take a message-list argument will default to the current message if no message-list is given. In the following descriptions, boldface denotes the name of a command, compose escape or option. Italics are used for arguments to commands or compose escapes. The vertical bar indicates selection and is used to separate the arguments from which you may select. All other text should be read literally.

RETURN	Displays the next message.
+n	[+] With no *n* argument, it displays the next message. If given a numeric argument *n*, goes to the *n*th message and displays it.
-n	[-] With no *n* argument, goes to the previous message and displays it. If given a numeric argument *n*, goes to the *n*th previous message and displays it.
^	Displays the first message.
$	Displays the last message.
=	Displays the message number of the current message.
?	Displays the summary of **mail** commands in */usr/lib/mail/mailhelp.cmd*.
!*shell-cmd*	Executes the shell command that follows. No space is needed after the exclamation point.

Alias *users* — Displays system-wide aliases for users. At least one user must be specified.

alias *name users* — [a] Aliases *users* to *name*. With no name arguments, displays all currently defined aliases. With one argument, displays the users aliased by the given name argument.

cd *directory* — [c] Changes the user's working directory to the specified directory. If no directory is given, then changes to the user's home directory.

delete *mesg-list* — [d] Deletes each message in the given message-list.

dp *mesg-list* — Deletes the current message and displays the next message.

echo *path* — Expands shell metacharacters.

edit *mesg-list* — [e] Takes the given message-list and points the text editor at each message in turn. On return to command mode, the edited message is read back in. See also the **visual** command.

exit[!] — [x] Immediately returns to the shell without modifying the system mailbox, the user mailbox, or a file specified with the **-f** switch.

file — [fi] Displays the name of the mailbox file.

forward *mesg-num user-list* — [f] Takes a *user-list* argument and forwards the current message to each name. The message sent to each is indented and shows that the sender has passed it on. The *mesg-num* argument is optional, and is used to forward the numbered message instead of the default message.

Forward *mesg-num user-list* — [F] Same as **forward** except that the message is not indented.

headers *+n | -n | mesg-list*
[**h**] With no argument, lists the current range of headers, which is an 18-message group. If a plus (+) argument is given, then the next 18-message group is displayed, and if a minus (-) argument is given, the previous 18-message group is displayed. Both plus and minus accept an optional numeric argument indicating the number of header-windows to move forward or backward. If a message-list is given, then the message-header for each message in the list is displayed.

help Same as ? above. Prints the summary of **mail** commands in */usr/lib/mail/mailhelp.cmd.*

hold *mesg-list* [**ho**] Takes a message-list and marks each message to be saved in the user's system mailbox instead of in *mbox.*

list Prints list of **mail** commands.

lpr *mesg-list* [**l**] Prints each of the messages in the required message-list on the lineprinter. Messages are piped through *pr* before being printed.

mail *[user-list]* [**m**] Takes an optional user-list argument and sends mail to each name after entering compose mode.

mbox *mesg-list* [**mb**] Marks messages given in the message-list argument to be saved in the user mailbox when a **quit** is executed. Message headers contain an initial letter "M" to show that they are to be saved.

move *mesg-list mesg-num*
Places the messages specified in *mesg-list* after the message specified in *mesg-num.* If *mesg-num* is 0, *mesg-list* moves to the top of the mailbox.

print *mesg-list* [p] Takes a message-list and displays each message on the user's terminal.

quit [q] Terminates the **mail** session, retaining all nondeleted, unsaved messages in the system mailbox. If the **autombox** option is set, then examined messages are saved in the user mailbox, deleted messages are discarded, and all messages marked with the **hold** command are retained in the system mailbox.

If you are executing a **quit** while editing a mailbox file with the **-f** flag, the mailbox file is rewritten and the user returns to the shell.

reply *mesg-list* [r] Takes a message-list and sends mail to each message author just like the **mail** command.

Reply *mesg-list* [R] Sends a reply to users named in the *To:* and *Cc:* fields, as well as the original sender.

restart Reads in mail that arrives during the current mail session.

save *mesg-list filename*
[s] Takes an optional message-list and a filename and appends each message in turn to
the end of the file. The default message is the current message.

set [se] Displays a list of available options.

set *option-list* [se] With no arguments, displays all variable values. Otherwise, sets option. Arguments are of the form *option=value*, if the option is a string option or just *option*, if the option is a switch. Multiple options may be set on one line.

shell [sh] Invokes an interactive version of the shell.

size *mesg-list* [si] Takes a message-list and displays the size in characters of each message.

source *file*
[so] Reads and executes **mail** commands from the named file.

string *string mesg-list*
Searches for *string* in *mesg-list*. If no *mesg-list* is specified, all undeleted messages are searched. Ignores case in search.

top
[t] Takes a message-list and displays the top five lines. The number of lines displayed is set by the variable *toplines*.

undelete *mesg-list*
[u] Takes a message-list and marks each one as not being deleted. Each message in the list must previously have been deleted.

unset *options*
[uns] Takes a list of option names and discards their remembered values; this is the opposite of **set**.

visual *mesg-list*
[v] Takes a message-list and invokes the **vi** editor on each one.

whois
Looks up a list of target mail recipients and prints the real names or descriptions of each recipient. If the first character of the first argument is alphabetic, the arguments are looked up without change. Otherwise, the arguments are assumed to be a message list, in the format specified in the *mail XENIX User's Guide*. For each message in the list, the "From" person is extracted from the header and added to list of users to be searched.

write *mesg-list filename*
[w] Writes the message bodies of messages given by the message-list to the file given by *filename*.

Compose Escape Summary

Compose escapes are used when composing messages to perform special functions. They are only recognized at the beginning of lines. The escape character can be set with the *escape* string option.(See the section "The escape String.") Abbreviations for each escape are in brackets.

Here is a summary of the compose escapes:

~~*string*	Inserts the string of text in the message prefaced by a single tilde (~).
~?	Prints out help for compose escapes on terminal.
~.	Same as Ctrl-d on a new line.
~!*command*	Executes a shell command, then returns to compose mode.
~|*command*	Pipes the message body through the command as a filter. Replaces the message body with the output of the filter. If the command gives no output or terminates abnormally, retains the original message body.
~_*mail-command*	Executes a **mail** command, then returns to compose mode.
~:*mail-command*	Executes a **mail** command, then returns to compose mode.
~**alias**	[~**a**] Displays a list of private aliases.
~**alias** *aliasname*	[~**a**] Displays the names included in private *aliasname*.
~**alias** *aliasname users*	[~**a**] Adds *users* to private *aliasname* list.
~**Alias**	[~**A**] Performs aliasing by first examining private aliases and then system-wide aliases using all three global alias files. Only the final result is printed (non-local mail recipients will have the complete delivery path printed). The user list is taken from header fields.

~Alias *users*	[**~A**] Performs aliasing by first examining private aliases and then system-wide aliases using all three global alias files. Only the final result is printed (non-local mail recipients will have the complete delivery path printed). At least one user must be specified.
~bcc *name ...*	[**~b**] Adds the given names to the *Bcc:* field.
~cc *name ...*	[**~c**] Adds the given name to the *cc:* field.
~dead	[**~d**] Reads the file *dead.letter* from your home directory into the message.
~editor	[**~e**] Invokes the line editor on the message being sent. Exiting the editor returns the user to compose mode.
~headers	[**~h**] Edits the message heading fields by printing each one in turn and allowing the user to modify each field.
~message *mesg-list*	[**~m**] Reads the named messages into the message being sent, shifted right one tab. If no messages are specified, reads the current message.
~Message *mesg-list*	[**~M**] Same as ~message except with no right shift.
~print	[**~p**] Prints the message buffer prefaced by the message heading.
~Print	[**~P**] Prints the real names or descriptions (in parentheses) after each recipient.
~quit	[**~q**] Aborts the message being sent, copying the message to *dead.letter* in your home directory if the *save* option is set.
~read *filename*	[**~r**] Reads the named file into the message.
~Return *name*	[**~R**] Adds the given names to the *Return-receipt-to:* field.
~shell	[**~sh**] Invokes a shell.

~subject *string* [~s] Causes the named string to become the current subject field.

~to *name ...* [~t] Adds the given names to the *To:* field.

~visual [~v] Invokes the **vi** editor to edit the message buffer. Exiting the editor returns the user to compose mode.

~write *filename* [~w] Writes the message body to the named file.

Option Summary

Options are controlled with the **set** and **unset** commands. An option is either a switch or a string. A switch is either on or off, while a string option has a value that is a pathname, a number, or a single character. Options are summarized below.

askcc Causes prompting for additional carbon copy recipients at the end of each message. Pressing RETURN retains the current list.

asksubject Causes prompting for the subject of each message you send. The subject is a line of text terminated by a RETURN.

autombox Usually messages are retained in the system mailbox when the user quits. However, if this option is **set**, examined messages are automatically appended to the user mailbox.

autoprint Causes the **delete** command to behave like **dp**. Thus, after deleting (or undeleting) a message, the next one is printed automatically.

chron Causes messages to be listed in chronological order.

dot Causes a single period on a newline to act as the EOT character. The normal end-of-transmission character, Ctrl-d, still works.

EDITOR= Pathname of the text editor to use in the **edit** command and ~e escape. If not defined, then a default editor is used.

escape=*char* If defined, sets *char* as the character to use in place of the tilde (~) to denote compose escapes.

ignore Causes interrupt signals from your terminal to be ignored and echoed as at-signs (@).

mchron Causes messages to be listed in numerical order (most recently received first), but displayed in chronological order.

metoo Normally, before sending, the name of the sender is removed from alias expansions. If *metoo* is set, then the name of the sender is *not* removed.

nosave Prevents saving of the message buffer in the file *dead.letter* in the home directory, after two consecutive interrupts or a ~q escape.

page=*n* Specifies the number of lines (*n*) to be printed in a "page" of text when displaying messages.

quiet Suppresses the printing of the version when **mail** is first invoked.

record= Sets the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is *not* copied.

SHELL= Pathname of the shell to use in the **!** command and the ~! escape. A default shell is used if this option is not defined.

toplines= Sets the number of lines of a message to be printed with the **top** command. Default is five lines.

verify Causes each target mail recipient to be verified. This option permits errors made while composing messages to be corrected or ignored.

VISUAL= Pathname of the text editor to use in the **visual** command and ~v escape. The default is for the **vi** editor.

Chapter 4

Communicating with Other Sites

Introduction

XENIX systems include a series of utilities that allow you to communicate with other computer sites. The particular utilities you use depend on how your computer is connected to the other site, what tasks you want to accomplish on the other site, and what operating system is running on the other site.

If the site is in close proximity to your computer, in the same room, for example, then it is likely that the two computers are connected by a simple serial line. If the site is a XENIX site, use the Micnet commands discussed in "Using Micnet" below to transfer files between the two sites and to execute commands on the remote site. If the site is a XENIX site, use the UUCP commands discussed in "Using UUCP" below.

If, on the other hand, the site you want to communicate with is on another floor, or across the country, your computer is connected to it by telephone lines. If the site is a XENIX or UNIX site, use the UUCP commands discussed in "Using UUCP" below to transfer files between the two sites and execute commands on the remote site. If the site is not a XENIX or UNIX site, use the commands discussed in "Using cu" below.

Neither the UUCP commands nor the Micnet commands allow you to have an *interactive* session with the remote site. If you want to have an interactive session, use the commands discussed in "Using cu" below.

This chapter assumes that your UUCP and/or Micnet networks are configured already. If this is not true, refer to "Building a Remote Network with UUCP" and "Building a Local Network with Micnet" in the *XENIX System Administrator's Guide* for more information.

Using Micnet

A Micnet network is a network of two or more computers connected by serial communication lines. A serial communication line is a cable with RS-232 connectors on each end.

The computers in a Micnet network use three commands to "talk" to one another. These are **rcp, remote** and **mail**. The **rcp** command is used to transfer files between machines in the network. The **remote** command is used to execute XENIX commands on a remote Micnet machine. The **mail** command is used to communicate with users on a remote computer. Each of these commands is discussed in the following sections.

Transferring Files with rcp

The **rcp** command is used to transfer copies of both text and binary files between machines connected in a Micnet network. Its syntax is similar to that of the **cp** command:

rcp [*options*] [*src_computer*:]*src_file* [*dest_computer*:]*dest_file*

These arguments mean the following:

src_file	The name of the file that you want to copy.
src_computer	The name of the computer on which *src_file* is located.
dest_file	The name of the copied file on the receiving computer. Usually, *src_file* and *dest_file* are the same.
dest_computer	The name of the computer on which *dest_file* is located.

You must have read permission on the source file and read and execute permissions on the directory that contains the source file in order to copy it with **rcp**. In addition, you must have write permission on the directory on the computer that is to receive the source file.

As an example, suppose you have three computers named *machine1*, *machine2* and *machine3* connected in a Micnet network. Suppose also that you want to send a copy of a file named *transfile* in the */usr/markt*

directory on *machine1* to the */tmp* directory on *machine3*. To do so, enter the following command:

rcp machine1:/usr/markt/transfile machine3:/tmp/transfile

If you are in the directory that contains the source file, specify the filename only. You do not have to specify the full machine and pathname. Using the example above, enter the following command from */usr/markt* on *machine1* to copy *transfile* to */tmp* on *machine3*:

rcp transfile machine3:/tmp/transfile

In addition to using **rcp** to send copies of files to remote computers, you can use **rcp** to retrieve copies of files from remote computers. Using the example above, suppose that *machine3* is your local computer and that you want to get a copy of */usr/markt/transfile* from *machine1*. To do so, enter the following command:

rcp machine1:/usr/markt/transfile /tmp/transfile

This command would place a copy of */usr/markt/transfile* on *machine1* in the */tmp* directory on *machine3*.

Because files are not sent immediately, an **rcp** transfer may take a few minutes. Files are copied to a spool directory and sent when the appropriate daemons "awaken." (A daemon is a program that periodically runs in the background.) In the case of **rcp**, the daemon that transfers files is the **daemon.mn** daemon.

rcp Options

Two options are available for use with **rcp**. These are **-m** and **-u** [machine:]*user*. The **-m** option causes mail to be sent to the user who entered the **rcp** command, reporting on the success or failure of the transfer. If you want **mail** to report to another user, use **-u** [machine:]*user*. This causes **mail** to report to *user* on *machine*.

The following command, issued from */usr/markt* on *machine1*, sends a copy of */usr/markt/transfile* on *machine1* to the */tmp* directory on *machine3*. Since the **-m** option is specified, mail will be sent reporting on the success or failure of the command:

rcp -m transfile machine3:/tmp/transfile

For more information on the **rcp** command, see **rcp**(C).

Executing Commands with remote

The **remote** command allows execution of commands across serial lines. The syntax of the **remote** command is:

remote [*options*] *site_name command* [*arguments*]

If the remote command produces output, that output is mailed to your system mailbox. Otherwise, **remote** sends mail only if the remote command fails to execute.

As an example, suppose that you are working on *machine1* and that you want to list the contents of the */tmp* directory on *machine2*. To do so, enter the following command:

remote machine2 ls /tmp

Since the **ls** command produces output, the output is mailed to you. In this case, your mail contains a listing of the contents of */tmp* on *machine2*.

remote Options

Two very useful options to the **remote** command are the **-m** and **-f***file* options. The **-m** option sends mail to you reporting on the success or failure of the command execution. Suppose, for example, that you want to remove */test* from */tmp/markt* on *machine2*. To do so, enter the following command:

remote -m machine2 rm /tmp/markt/test

After this command is executed, you receive mail reporting on the success or failure of the **rm** command.

The **-f***file* option allows you to specify a file on the local computer that contains the input for the command that is to be executed on the remote computer. As an example, suppose that you have a file named *chapter1* on your local computer that you want to print on *machine2's* default printer. To do so, enter the following command:

remote -m -f chapter1 machine2 lp

Because the **-m** option is specified, you are informed by mail of the success or failure of the **remote** command.

Note

The system administrator can specify which commands are allowed to execute remotely over serial lines on which computers. The commands that are allowed to execute remotely on a XENIX system are listed in the computer's */etc/default/micnet* file. Any XENIX command can execute remotely if the computer's */etc/default/micnet* file contains the statement *executeall* on a line by itself.

Transferring Files with mail

The **mail** command can be used to transfer files between computers in a Micnet network. However, there are several drawbacks to using **mail** for this purpose:

- You must transfer the file to a *user* on the remote system, rather than to a *directory*.
- You can only use **mail** to transfer small files. Large files are randomly truncated by **mail**.
- You cannot transfer binary files with **mail**.

On the other hand, **mail** is very useful for sending small files to several users at once on a remote system. For information on using **mail**, see "mail" in this guide.

Using UUCP

UUCP is a series of programs that provide networking capabilities for XENIX systems. While UUCP commands can be used over serial lines, they are usually used on computers connected by telephone lines.

The UUCP programs allow you to transfer files between remote computers and to execute commands on remote computers. Since the computers may be connected by telephone lines, UUCP transfers can take place over thousands of miles. A UUCP site in New York City can transfer a file to or execute a command on a connected UUCP site in San Francisco, or Jakarta, or anywhere in the world. The following sections explain how to use these UUCP programs.

Transferring Files with uucp

Both the **uucp** and **uuto** commands can be used to transfer copies of binary and text files between remote UUCP sites. There are advantages and disadvantages to each. The **uucp** command gives you great flexibility in specifying where on the remote system the transferred file is to be placed. However, **uucp** syntax can be rather long and complicated. The **uuto** command, on the other hand, is easy to use. But **uuto** restricts where you can place the file on the remote system. In addition, retrieving a file sent with **uuto** is slightly more complicated than retrieving a file sent with **uucp**.

The **uucp** command is discussed in this section. The **uuto** command is discussed in the following section.

Before You Begin

Before you can copy files to remote sites with **uucp**, you must verify that:

- Your local site is a "dial out" site.
- Your local site "knows" how to call the remote site.
- The files that you want to send have read permission set for others.
- The directory that contains the file that you want to send has read and execute permissions set for others.

- Your computer has write permission in the directory on the remote site to which you want to copy the file.

Each of these is discussed below.

Some UUCP sites are "dial-in" sites, some are "dial-out" sites, and some are both. Verify that your site is a dial-out site. If it is not, your computer might have the capability to be on the receiving end of a UUCP connection, but not on the calling end.

You must be sure that your computer "talks" to the site with which you want to communicate. The **uuname** command gives you this information. Entering **uuname** with no options lists the UUCP sites your computer talks to directly. Entering **uuname** with the **-l** option causes the name of your computer to be displayed.

Note that you may be able to communicate with a site that does not show up in a **uuname** listing. This is possible because UUCP sites are often "chained together." So if you know that a site you want to transfer files to communicates with a site that your system communicates with, you can send files to the first site through the second. An example is provided below under "Indirect Transfers."

In order to copy a file to a remote UUCP site, the file must have read permission set for others and the directory that contains the file must have read and execute permissions set for others. Use the **l** command to examine the file's permissions and the **l -d** command to examine the directory's permissions. If the permissions are not correct, enter the following commands to set the correct permissions:

chmod o+r *filename*
chmod o+rx *directory*

Finally, you must verify that your computer has write permission on the directory on the remote site to which you want to transfer files. Each remote UUCP site has a */usr/lib/uucp/Permissions* file. This file specifies the directories on that site from which your computer can read and to which your computer can write. You can only send a file to a directory on a remote site if your computer has write permissions on that directory, as specified on the remote site's */usr/lib/uucp/Permissions* file.

By default, most UUCP sites permit calling-in computers to write to their */usr/spool/uucppublic* directory. Since there is no way to find out which directories your computer can write to on the remote site, short of contacting somebody at the site, the safest thing to do when making a UUCP transfer is to write to */usr/spool/uucppublic*. The procedure for doing this is outlined below.

Using uucp

The syntax of the **uucp** command is similar to the syntax of **cp**:

uucp [*options*] *src_computer!src_file dest_computer!dest_file*

These arguments mean the following:

src_file	The name of the file that you want to copy.
src_computer	The name of the computer on which *src_file* is located.
dest_file	The name of the copied file on the receiving computer. Usually, *src_file* and *dest_file* are the same.
dest_computer	The name of the computer on which *dest_file* is located.

There are several different ways to specify the location on the remote machine to which you want to transfer the file. The simplest is the *~/dest_file* specification. This is also the safest specification, because *~/dest_file* is expanded to */usr/spool/uucppublic/dest_file*, thereby assuring that the transfer will succeed.

For example, to send */usr/markt/transfile* on *machine1* to */usr/spool/uucppublic* on *machine2*, enter the following command:

uucp /usr/markt/transfile machine2!~/transfile

This command creates the file */usr/spool/uucppublic/transfile* on *machine2*.

If */usr/markt* is your current directory, you can copy *transfile* to *machine2* with the following command:

uucp transfile machine2!~/transfile

The **uucp** command works much like the **rcp** command. Files are not copied and sent immediately. Instead, copies are placed in a spool directory and sent once the appropriate daemon awakens. In the case of the UUCP programs, the daemon is the **uucico** daemon. Depending on how your system is configured, a **uucp** transfer might take place within minutes, or it might take hours.

Note

Because the exclamation mark has special meaning to the C-shell, you must "escape" with a backslash (\) any exclamation marks that appear in a **uucp** command, if you are using the C-shell. For a C-shell user, the command above is specified as:

uucp transfile machine2\!~/transfile

Another form of the command allows you to specify the full pathname of the copied file on the remote computer. This is for sending the file to a specific directory on the remote system. However, you must be sure that your computer has write permission on this directory, otherwise the transfer will fail.

As an example, suppose that you want to send *transfile* in */usr/markt* on *machine1* to the */usr/cindy* directory *machine2*. To do so, enter the following command:

uucp /usr/markt/transfile machine2!/usr/cindy/transfile

Note that, like the **rcp** command, the **uucp** command can be used to retrieve files from a remote site, in addition to copying files to a remote site. Using the example above, if your local computer is *machine2* and you want to send a copy of */usr/markt/transfile* on *machine1* to the */usr/cindy* directory on *machine2*, enter the following command:

uucp machine1!/usr/markt/transfile /usr/cindy/transfile

You can also use *~user* to specify a location on the remote computer. The *~user* argument is expanded to the pathname of the home directory of the person on the remote computer whose login is *user*. For example, if */usr/cindy* is the home directory of a user whose login is *cindy* on *machine2*, enter the following command from the */usr/markt* directory on *machine1* to copy */usr/markt/transfile* to */usr/cindy*:

uucp transfile machine2!~cindy/transfile

The receiving computer expands *~cindy* to the full pathname of *cindy's* home directory, creating */usr/cindy/transfile*. Again, your computer must have write permission in *cindy's* home directory in order for this transfer to succeed.

Indirect Transfers

You might be able to send files to a UUCP site not listed in a **uuname** listing. As an example, suppose that your local computer is connected to a UUCP site named *machine2*. Suppose also that *machine2* is connected to a UUCP site named *machine3*. You can send */tmp/transfile* on your local computer to */usr/spool/uucppublic* on *machine3*. Do so by specifying the full UUCP address relative to your local computer:

```
uucp /tmp/transfile machine2!machine3!~/transfile
```

Note that each site name in the command line is followed by an exclamation mark. By placing several site names in a **uucp** command line, you can greatly extend the range of systems to which you can copy files with **uucp**. This is also true for the **uuto** and **uux** commands discussed below.

uucp Options

Several options are available for the **uucp** command. Some of the most useful are the -m and **-n** *user* options.

The **-m** option sends you mail reporting on the success or failure of the file transfer. The **-n** *user* option notifies the *user* on the machine to whom the files are sent of the file transfer.

Other options are available for use with **uucp**. Refer to **uucp**(C) for a complete list of these options.

Checking the Status with uustat

You can use the **uustat** command to check on the status of files you copied with **uucp**. To check on the status of all your **uucp** jobs, enter the following command:

```
uustat
```

Your output looks like the following:

```
1234 markt machine2 2/19-10:29 2/19-10:40 JOB IS QUEUED
```

Reading from left to right, the elements of this message are:

1234	This is the job number assigned to this **uucp** transfer.
markt	This is the user who requested the transfer.
machine2	This is the site name of the recipient's computer.
2/19-10:29	This is the date and time the job was queued in the spool directory.
2/19-10:40	This is the date and time of the **uustat** request.
Job Status	This message tells you the status of the job. In this case, JOB IS QUEUED tells you that the job is in the spool directory waiting to be sent. When the transfer is completed, **uustat** displays the message: COPY FINISHED, JOB DELETED

Several options are available for use with **uustat**. Refer to **uustat**(C) for more information.

Transferring Files with uuto

The **uuto** command allows you to copy files to the public directory of a UUCP site to which your system is connected. The public directory on most XENIX and UNIX systems is */usr/spool/uucppublic*. The syntax of **uuto** is:

uuto [*options*] *source_file destination_computer!login*

The *login* argument is the login of the user to whom you are sending files.

Before you can send a file with **uuto**, you must verify that:

- The file has read permission set for others.
- The directory that contains the file has read and execute permissions set for others.

If the permissions are not correct, enter the following commands to set the correct permissions:

chmod o+r *filename*
chmod o+rx *directory*

Files sent with **uuto** are placed in the directory:

/usr/spool/uucppublic/receive/login/source_computer

In this example, *login* is the login of the user to whom you are sending files and *source_computer* is the site name of *your* system.

As an example, suppose that you want to send a copy of *transfile* in */tmp* on your computer, *machine1*, to a user whose login is *cindy* on *machine2*. To do so, enter the following command:

uuto /tmp/transfile machine2!cindy

This command copies *transfile* to the following directory:

usr/spool/uucppublic/receive/cindy/machine1

When the file transfer is complete, the recipient is notified by **mail** that the file has arrived. If the **-m** option is used on the **uuto** command line, the sender is notified by **mail** of the success or failure of the transfer.

Like **uucp**, files transferred with **uuto** are not transferred immediately after the command is entered. Instead, they are placed in a spool directory and sent when the **uucico** daemon awakens.

Retrieving Files with uupick

In order to retrieve a file sent by **uuto**, you must use the **uupick** command. To execute **uupick**, enter the following command:

uupick

The **uupick** program searches the public directory for any files sent to you. If it finds any, it responds with the following prompt:

```
from source_computer: file filename ?
```

The *source_computer* is the name of the sender's computer and *filename* is the name of the file transferred. In the example above, if the **uuto** transfer to *cindy* on *machine2* is successful, cindy sees the following **uupick** prompt:

```
from machine1: file transfile ?
```

Several options are available for responding to the **uupick** prompt. Two of the most useful are **m** [*dir*] and **d**. The **m** [*dir*] option tells **uupick** to move the file to directory *dir*. Once in *dir*, you can manipulate the file as you would any other file on your system. In the example above, cindy could enter the following in response to the **uupick** prompt:

m $HOME

This causes *transfile* to be moved from the public directory to cindy's home directory. If no directory is specified after **m**, the file is moved to the recipient's current directory.

Entering **d** at the **uupick** prompt causes the file to be deleted from the public directory. You can quit **uupick** by entering **q**. Note other **uupick** options are available. Refer to **uupick**(C) for a complete list of these.

Executing Commands with uux

The **uux** command is used to execute commands on remote UUCP sites and on files gathered from remote UUCP sites. For security reasons, the commands available for remote execution on a computer are often very limited. A computer's */usr/lib/uucp/Permissions* file lists the commands that can be executed remotely on that computer. If you attempt to execute a command not listed in this file, you will receive mail indicating that the command cannot be executed on the computer in question.

The syntax of **uux** is:

uux [*options*] *command-line*

The *command-line* argument looks like any other XENIX command line, with the exception that commands and filenames may be prefixed with *site-name!*.

The following is an example of how to execute a command on a remote system. The command causes */tmp/printfile* on *machine2* to be sent to *machine2's* default printer:

uux machine2!lp machine2!/tmp/printfile

Note that prefixing a site name to a command causes the command to be executed on that site.

The following is an example of how to execute a command on a local system on files gathered with **uux** from remote systems. Suppose that your local computer is connected to both *machine2* and *machine3*. Suppose also that you want to compare the contents of */tmp/chpt1* on *machine2* with */tmp/chpt1* on *machine3*. To do so, enter the following command:

uux "diff machine2!/tmp/chpt1 machine3!/tmp/chpt1 > diff.file"

This command will compare the contents of the files on *machine2* and *machine3* and place the output in *diff.file* in the current directory on the local computer. Since there is no site name prefixed to the **diff** command, the command is executed locally.

Note that, in the example above, the **uux** command line is placed in quotation marks. This is because it contains the redirect symbol (>). In general, place the **uux** command line in quotation marks whenever the command line contains special shell characters such as <, >, |, and so forth.

Logging in to Remote Systems

The **ct** command connects your system to a remote terminal with a modem attached. The **cu** command connects your system to a remote system. The remote system can be attached via phone lines or via a simple serial line. These commands differ from the Micnet commands and the UUCP commands discussed above in that your session with the remote system is *interactive*. The remote system "sees" you as just another user on the system. Both **ct** and **cu** are discussed below.

Using ct

The **ct** command connects a local computer to a remote terminal equipped with a modem and allows a user on that terminal to log in to the computer. To do this, the command dials the phone number of the remote modem. The remote modem must be able to answer the call automatically. When **ct** detects that the call has been answered, it issues a **getty** (login) process for the remote terminal and allows a user on the terminal to log in on the computer.

This command is especially useful when issued from the opposite end, that is, from the remote terminal itself. If you are using a remote terminal and you want to avoid long distance charges, you can use **ct** to have the computer place a call to your terminal. To do so, simply call the computer, log in, and issue the **ct** command. The computer will hang up the line and call your 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. If you answer no, **ct** quits.

The syntax of **ct** is:

ct [*options*] *telno*

The argument *telno* is the telephone number of the remote terminal.

As an example, suppose that you have a terminal with a modem attached at home and that you want to log in to the computer at work from this terminal. To avoid long distance charges, first call your work computer and log in. Then issue the **ct** command to make the computer hang up and call your terminal back. If your phone number is 932-3497, the **ct** command is:

```
ct -s1200 9323497
```

The **-s** option tells **ct** to call the modem at 1200 baud. If no device is available on the computer at work, you see the following message after executing **ct**:

```
The one 1200 baud dialer is busy
Do you want to wait for dialer? (y for yes):
```


If you type **n** (no), the **ct** command exits. If you type **y** (yes), **ct** prompts you to specify how long **ct** should wait:

```
Time, in minutes?
```

If a dialer is available when you enter the **ct** command, you see the following message:

```
Allocated dialer at 1200 baud
```

This means that a dialer has been found. You are then asked if you want the line connecting your remote terminal to the computer to be dropped:

```
Proceed to hang-up? (y to hang-up, otherwise exit):
```

Since you want to avoid long-distance charges by having the computer call you, answer **y** (yes). You are then logged off and **ct** calls your remote terminal back.

As another example, suppose that you are logged in on a computer through a local terminal and that you want to connect a remote terminal

to the computer. The phone number of the modem on the remote terminal is 932-3497. To connect the terminal, enter the following command:

```
nohup ct -h -s1200 9323497 &
```

The **-h** option tells **ct** not to disconnect the local terminal (the terminal on which the command was issued) from the computer. After the command is executed, a login prompt is displayed on the remote terminal. The user can then log in and work on the computer just as on a local terminal.

Several options are available for **ct**. Refer to **ct**(C) for a complete list of these options.

Using cu

The **cu** command connects your local computer to a remote computer and allows you to be logged in on both computers simultaneously. The remote computer does not have to be a XENIX system.

If the remote computer is a XENIX system, **cu** allows you to move back and forth between the two computers, transferring files and executing commands on both. Note that **cu** only allows you to transfer text files. You cannot transfer binary files with **cu**. To transfer binary files to a remote XENIX system, use either **rcp** or **uucp**.

The syntax of the **cu** command is:

cu [*options*] *target*

The *target* argument can take one of three forms:

phone number — This is the number of the remote computer to which you want to connect. You can embed equal signs, which represent secondary dial tones, and dashes, which represent four-second delays, in the phone number. A sample phone number might be **4084551222--341**. This number contains an area code and number, two dashes for an eight second delay and an extension.

system-name	This is the name of a system that is listed in the */usr/lib/uucp/Systems* file. The **cu** command obtains the telephone number and the baud rate of *system-name* from this file. The **-s**, **-n**, and **-l** options should not be used with *system-name*. To see the list of computers in the *Systems* file, enter: **uuname**.
-l *line*	This is the device name of the serial line connected to the remote computer. It has the form *ttyXX*, where *XX* is the number of a serial line.
-l *line dir*	Connects directly with serial line instead of making a phone connection.

Several options are available for use with the **cu** command. Refer to **cu**(C) for a complete list of these options.

Once the connection is made, if the remote computer is a XENIX system, you are presented with a login prompt. Log in as you would if you were connected locally. When you finish working on the remote computer, log off as you would if you were connected locally. Then terminate the **cu** connection by entering a tilde followed by a period (~.). You are still logged in on the local computer.

As an example, suppose that you want to log in to a remote XENIX computer via the phone lines. Suppose also that the remote computer's number is 847-7867. To connect to the remote computer, enter the following command:

cu -s1200 8477867

The **-s1200** option causes **cu** to use a 1200 baud dialer. If the **-s** option is not specified, **cu** uses the first available dialer at the speed specified in the *Devices* file.

When the remote XENIX system answers the call, **cu** notifies you that the connection has been made by displaying the following message:

```
Connected
```

Next, you are prompted for your login:

```
login:
```

Enter your login and password. Once you enter this information, you can use this computer as if you were logged in locally. When you are finished, logout and then enter:

```
~.
```

This terminates the **cu** session.

cu Command Strings

Several "Command Strings" are available with **cu** that allow your local computer to communicate with a remote XENIX system. Two of the most useful are **take** and **put**.

The **take** command allows you to copy files from the remote computer to the local computer. Suppose, for example, that you want to copy a file named *proposal* in the current directory of the remote computer to your home directory on the local computer. To do so, enter the following command:

```
~%take proposal $home/proposal
```

Note that you have to prefix a tilde and a percent sign (**~%**) to the **take** command, and that the tilde must be placed at the start of a line. For this reason, it is a good idea to press ⟨Return⟩ before using **take**.

The **put** command allows you to do the opposite of **take**. It copies files from the local computer to the remote computer. Suppose, for example, that you want to copy a file named *minutes* from your home directory on the local computer to the */tmp* directory of the remote computer. Suppose

also that you want the file to be called *minutes.9-18* on the remote computer. To do so, enter the following command:

~%put $home/minutes /tmp/minutes.9-18

Like the **take** command, you have to prefix a tilde and a percent sign (**~%**) to the **put** command, with the tilde coming at the beginning of a line. Note also that **take** and **put** copy only text files, and only to XENIX systems. They do not copy binary files.

Note

The **cu** command cannot detect or correct transmission errors. After a file transfer, you can check for loss of data by running the **sum** command on both the file that was sent and the file that was received. This command reports the total number of bytes in each file. If the totals match, your transfer was probably successful. See the **sum**(C) manual page for details.

Other command strings are available for use with **cu**. For a complete list of these, see **cu**(C).

Chapter 5

The Shell

Introduction

When users log into a XENIX system, they communicate with one of several interpreters. This chapter discusses the shell command interpreter, **sh**. This interpreter is a XENIX program that supports a very powerful command language. Each invocation of this interpreter is called a shell; and each shell has one function: to read and execute commands from its standard input.

Because the shell gives the user a high-level language in which to communicate with the operating system, you can perform tasks unheard of in less sophisticated operating systems. Commands that would normally have to be written in a traditional programming language can be written with just a few lines in a shell procedure. In other operating systems, commands are executed in strict sequence. With the shell, commands can be:

- Combined to form new commands
- Passed positional parameters
- Added or renamed by the user
- Executed within loops or executed conditionally
- Created for local execution without fear of name conflict with other user commands
- Executed in the background without interrupting a session at a terminal

Furthermore, commands can "redirect" command input from one source to another and redirect command output to a file, terminal, printer, or to another command. This provides flexibility in tailoring a task for a particular purpose.

Basic Concepts

The shell itself (that is, the program that reads your commands when you log in or that is invoked with the **sh** command) is a program written in the C language; it is not part of the operating system proper, but an ordinary user program.

How Shells Are Created

On a XENIX system, a process is an executing entity complete with instructions, data, input, and output. All processes have lives of their own, and may even start (or "fork") new processes. Thus, at any given moment several processes may be executing, some of which are "children" of other processes.

Users log into the operating system and are assigned a "shell" from which they execute. This shell is a personal copy of the shell command interpreter that is reading commands from the keyboard: in this context, the shell is simply another process.

In the XENIX multitasking environment, files may be created in one phase and then sent off to be processed in the "background." This allows the user to continue working while programs are running.

Commands

The most common way of using the shell is by entering simple commands at your keyboard. A *simple command* is any sequence of arguments separated by spaces or tabs. The first argument (numbered zero) specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are passed as arguments to that command. For example, the following command line might be entered to request printing of the files *allan*, *barry*, and *calvin*:

```
lpr allan barry calvin
```

If the first argument of a command names a file that is *executable* (as indicated by an appropriate set of permission bits associated with that file) and is actually a compiled program, the shell, as parent, creates a child process that immediately executes that program. If the file is marked as being executable, but is not a compiled program, it is assumed

to be a shell procedure, that is, a file of ordinary text containing shell command lines. In this case, the shell spawns another instance of itself (a *subshell*) to read the file and execute the commands inside it.

From the user's viewpoint, compiled programs and shell procedures are invoked in exactly the same way. The shell determines which implementation has been used, rather than requiring the user to do so. This provides uniformity of invocation.

How the Shell Finds Commands

The shell normally searches for commands in three distinct locations in the file system. The shell attempts to use the command name as given; if this fails, it prepends the string */bin* to the name. If the latter is unsuccessful, it prepends */usr/bin* to the command name. The effect is to search, in order, the current directory, then the directory */bin*, and finally, */usr/bin*. For example, the **pr** and **man** commands are actually the files */bin/pr* and */usr/bin/man*, respectively. A more complex pathname may be given, either to locate a file relative to the user's current directory, or to access a command with an absolute pathname. If a given command name includes a slash (/) (for example, */bin/sort dir/cmd*), the prepending is not performed. Instead, a single attempt is made to execute the command as named.

This mechanism gives the user a convenient way to execute public commands and commands in or near the current directory, as well as the ability to execute any accessible command, regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command does not affect a user who already has a private command with the same name. The particular sequence of directories searched may be changed by resetting the shell PATH variable. (Shell variables are discussed later in this chapter.)

Generation of Argument Lists

The arguments to commands are very often filenames. Sometimes, these filenames have similar, but not identical, names. To take advantage of this similarity in names, the shell lets the user specify patterns that match the filenames in a directory. If a pattern is matched by one or more filenames in a directory, then those filenames are automatically generated by the shell as arguments to the command.

Most characters in such a pattern match themselves, but there are also XENIX special characters that may be included in a pattern. These special characters are: the star (*), which matches any string, including the null string; the question mark (?), which matches any one character; and any sequence of characters enclosed within brackets ([and]), which matches any one of the enclosed characters. Inside brackets, a pair of characters separated by a dash (-) matches any character within the range of that pair. Thus [a-de] is equivalent to [abcde].

Examples of metacharacter usage:

Metacharacter	Meaning
*	Matches all names in the current directory
temp	Matches all names containing *"temp"*
[a-f]*	
*.c	
/usr/bin/?	Matches all single-character names in */usr/bin*

This pattern-matching capability saves typing and, more importantly, makes it possible to organize information in large collections of files that are named in a structured fashion, using common characters or extensions to identify related files.

Pattern matching has some restrictions. If the first character of a filename is a period (.), it can be matched only by an argument that literally begins with a period. If a pattern does not match any filenames, then the pattern itself is the result of the match.

Note that directory names should not contain any of the following characters:

```
* ? [ ]
```

If these characters are used, then infinite recursion may occur during pattern matching attempts.

Quoting Mechanisms

Several characters, including <,>,*,?,[and], have special meanings to the shell. To remove the special meaning of these characters requires some form of quoting. This is done by using single quotation marks (') or double quotation marks (") to surround a string. A backslash (\) before a single character provides this function. (Back quotation marks (`) are used only for command substitution in the shell and do not hide the special meanings of any characters.)

All characters within single quotation marks are taken literally. Thus:

```
echostuff='echo $? $*; ls *| wc'
```

results in the string:

```
echo $? $*; ls *| wc
```

being assigned to the variable *echostuff*, but it does *not* result in any other commands being executed.

Within double quotation marks, the special meaning of certain characters does persist, while all other characters are taken literally. The characters that retain their special meaning are the dollar sign ($), the backslash (\), the back quotation mark (`), and the double quotation mark (") itself. Thus, within double quotation marks, variables are expanded and command substitution takes place (both topics are discussed in later sections). However, any commands in a command substitution are unaffected by double quotation marks, so that characters such as star (*) retain their special meaning.

To hide the special meaning of the dollar sign ($) and single and double quotation marks within double quotation marks, precede these characters with a backslash (\). Outside of double quotation marks, preceding a character with a backslash is equivalent to placing single quotation marks around that character. A backslash (\) followed by a newline causes that newline to be ignored. The backslash-newline pair is therefore useful in allowing continuation of long command lines.

Some examples of quoting are displayed below:

Input	Shell interprets as:
'`'	The back quotation mark (`)
'"'	The double quotation mark (")
``echo one``	the one word "`echo one`"
"\""	The double quotation mark (")
"`echo one`"	the one word "one"
"`"	illegal (expects another `)
one two	the two words "one" & "two"
"one two"	the one word "one two"
'one two'	the one word "one two"
'one * two'	the one word "one * two"
"one * two"	the one word "one * two"
`echo one`	the one word "one"

Standard Input and Output

In general, most commands do not know or care whether their input or output is coming from or going to a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline. A few commands vary their actions depending on the nature of their input or output, either for efficiency, or to avoid useless actions (such as attempting random access I/O on a terminal or a pipe).

When a command begins execution, it usually expects that three files are already open: a "standard input" , a "standard output" , and a "diagnostic output" (also called "standard error"). A number called a *file descriptor* is associated with each of these files. By convention, file descriptor 0 is associated with the standard input, file descriptor 1 with the standard output, and file descriptor 2 with the diagnostic output. A child process normally inherits these files from its parent; all three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the terminal screen). The shell permits the files to be redirected elsewhere before control is passed to an invoked command.

An argument to the shell of the form "*<file*" or "*>file*" opens the specified file as the standard input or output (in the case of output, destroying the previous contents of *file*, if any). An argument of the form "*>>file*" directs the standard output to the end of *file*, thus providing a way to append data to the file without destroying its existing contents. In either of the two output cases, the shell creates *file* if it does not already exist. Thus:

```
> output
```

alone on a line creates a zero-length file. The following appends to file *log* the list of users who are currently logged on:

```
who >> log
```

Such redirection arguments are only subject to variable and command substitution; neither blank interpretation nor pattern matching of filenames occurs after these substitutions. This means that:

```
echo 'this is a test' > *.gal
```

produces a one-line file named **.gal*. Similarly, an error message is produced by the following command, unless you have a file with the name "?":

```
cat < ?
```

Special characters are *not* expanded in redirection arguments because redirection arguments are scanned by the shell *before* pattern recognition and expansion takes place.

Diagnostic and Other Outputs

Diagnostic output from XENIX commands is normally directed to the file associated with file descriptor 2. (There is often a need for an error output file that is different from standard output so that error messages do not get lost down pipelines.) You can redirect this error output to a file by immediately prepending the number of the file descriptor (2 in this case) to either output redirection symbol (> or >>). The following line appends error messages from the **cc** command to the file named *ERRORS*:

```
cc testfile.c 2>> ERRORS
```

Note that the file descriptor number must be prepended to the redirection symbol *without* any intervening spaces or tabs; otherwise, the number will be passed as an argument to the command.

This method may be generalized to allow redirection of output associated with any of the first ten file descriptors (numbered 0-9). For instance, if *cmd* puts output on file descriptor 9, then the following line will direct that output to the file *savedata*:

```
cmd 9> savedata
```

A command often generates standard output and error output, and might even have some other output, perhaps a data file. In this case, one can redirect independently all the different outputs. Suppose, for example, that *cmd* directs its standard output to file descriptor 1, its error output to file descriptor 2, and builds a data file on file descriptor 9. The following would direct each of these three outputs to a different file:

```
cmd >standard   2> error   9> data
```

Command Lines and Pipelines

A sequence of commands separated by the vertical bar (|) makes up a *pipeline*. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbors by *pipes*, that is, the output of each command (except the last one) becomes the input of the next command in line.

A *filter* is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read large amounts of data before producing output; **sort** is an example of the extreme case that requires all input to be read before any output is produced. The following is an example of a typical pipeline:

```
nroff -mm text | col | lpr
```

nroff is a text formatter available in the UNIX Text Processing System whose output may contain reverse line motions, **col** converts these motions to a form that can be printed on a terminal lacking reverse-motion capability, and **lpr** does the actual printing. The flag **-mm** indicates one of the commonly used formatting options, and *text* is the name of the file to be formatted.

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the ways described above. It may be helpful to try these at a terminal:

- **who**
  Prints the list of logged-in users on the terminal screen.

- **who >>log**
  Appends the list of logged-in users to the end of file *log*.

- **who | wc -l**
  Prints the number of logged-in users. (The argument to **wc** is pronounced "minus ell".)

- **who | pr**
  Prints a paginated list of logged-in users.

- **who | sort**
  Prints an alphabetized list of logged-in users.

- **who | grep bob**
  Prints the list of logged-in users whose login names contain the string *bob*.

- **who | grep bob | sort | pr**
  Prints an alphabetized, paginated list of logged-in users whose login names contain the string *bob*.

- **{ date; who | wc -l ; } >> log**
  Appends (to file *log*) the current date followed by the count of logged-in users. Be sure to place a space after the left brace and a semicolon before the right brace.

- **who | sed -e 's/ .*//' | sort | uniq -d**
  Prints only the login names of all users who are logged in more than once. Note the use of **sed** as a filter to remove characters trailing the login name from each line. (The ".*" in the **sed** command is preceded by a space.)

The **who** command does not *by itself* provide options to yield all these results—they are obtained by combining **who** with other commands. Note that **who** just serves as the data source in these examples. As an exercise, replace "who | " with "</etc/passwd" in the above examples to see how a file can be used as a data source in the same way. Notice that redirection arguments may appear anywhere on the command line, even at the start. This means that:

```
< infile >outfile sort | pr
```

is the same as:

```
sort < infile | pr > outfile
```


Command Substitution

Any command line can be placed within back quotation marks (`...`) so that the output of the command replaces the quoted command line itself. This concept is known as *command substitution*. The command or commands enclosed between back quotation marks are first executed by the shell and then their output replaces the whole expression, back quotation marks and all. This feature is often used to assign to shell variables. (Shell variables are described in the next section.)

For example:

```
today=`date`
```

assigns the string representing the current date to the variable "today"; for example "Tue Nov 26 16:01:09 EST 1985". The following command saves the number of logged-in users in the shell variable *users*:

```
users=`who | wc -l`
```

Any command that writes to the standard output can be enclosed in back quotation marks. Back quotation marks may be nested, but the inside sets must be escaped with backslashes (\). For example:

```
logmsg=`echo Your login directory is \`pwd\``
```

will display the line "your login directory is *name of login directory*". Shell variables can also be given values indirectly by using the **read** and **line** commands. The **read** command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named.

For example:

```
read first init last
```

takes an input line of the form:

```
G. A. Snyder
```

and has the same effect as entering:

```
first=G. init=A. last=Snyder
```

The **read** command assigns any excess "words" to the last variable.

The **line** command reads a line of input from the standard input and then echoes it to the standard output.

Shell Variables

The shell has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are referred to as *positional parameters*; these are the variables that are normally set only on the command line. Other shell variables are simply names to which the user or the shell itself may assign string values.

Positional Parameters

When a shell procedure is invoked, the shell implicitly creates *positional parameters*. The name of the shell procedure itself in position zero on the command line is assigned to the positional parameter $0. The first command argument is called $1, and so on. The **shift** command may be used to access arguments in positions numbered higher than nine. For example, the following shell script might be used to cycle through command line switches and then process all succeeding files:

```
while test -n "$1"
      do case $1 in
            -a) A=aoption ; shift ;;
            -b) B=boption ; shift ;;
            -c) C=coption ; shift ;;
      -*) echo "bad option" ; exit 1 ;;
      *) process rest of files
      esac
done
```

One can explicitly force values into these positional parameters by using the **set** command. For example:

```
set abc def ghi
```

assigns the string "abc" to the first positional parameter, $1, the string "def" to $2, and the string "ghi" to $3. Note that $0 may not be assigned a value in this way—it always refers to the name of the shell procedure; or in the login shell, to the name of the shell.

User-Defined Variables

The shell also recognizes alphanumeric variables to which string values may be assigned. A simple assignment has the syntax:

name=string

Thereafter, *$name* will yield the value *string*. A *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore. No spaces surround the equal sign (=) in an assignment statement. Note that positional parameters may not appear on the left side of an assignment statement; they can only be set as described in the previous section.

More than one assignment may appear in an assignment statement, but beware: *the shell performs the assignments from right to left.* Thus, the following command line results in the variable "A" acquiring the value "abc":

```
A=$B B=abc
```

The following are examples of simple assignments. Double quotation marks around the right-hand side allow spaces, tabs, semicolons, and newlines to be included in a string, while also allowing variable substitution (also known as "parameter substitution") to occur. This means that references to positional parameters and other variable names that are prefixed by a dollar sign ($) are replaced by the corresponding values, if any. Single quotation marks inhibit variable substitution:

```
MAIL=/usr/mail/gas
echovar="echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

In the above example, the variable *echovar* has as its value the string consisting of the values of the first four positional parameters, separated by spaces, plus the string "echo". No quotation marks are needed around the string of asterisks being assigned to *stars* because pattern matching (expansion of star, the question mark, and brackets) does not apply in this context. Note that the value of *$asterisks* is the literal string "$stars", *not* the string "*****", because the single quotation marks inhibit substitution.

In assignments, spaces are not re-interpreted after variable substitution, so that the following example results in *$first* and *$second* having the same value:

```
first='a string with embedded spaces'
second=$first
```

In accessing the values of variables, you may enclose the variable name in braces {...} to delimit the variable name from any following string. In particular, if the character immediately following the name is a letter, digit, or underscore, then the braces are required. For example, examine the following input:

```
a='This is a string'
echo "${a}ent test of variables."
```

Here, the **echo** command prints:

```
This is a stringent test of variables.
```

If no braces were used, the shell would substitute a null value for "$aent" and print:

```
test of variables.
```

The following variables are maintained by the shell. Some of them are set by the shell, and all of them can be reset by the user:

HOME — Initialized by the **login** program to the name of the user's *login directory*, that is, the directory that becomes the current directory upon completion of a login; **cd** without arguments switches to the $HOME directory. Using this variable helps keep full pathnames out of shell procedures. This is of great benefit when pathnames are changed, either to balance disk loads or to reflect administrative changes.

IFS — The variable that specifies which characters are *internal field separators*. These are the characters the shell uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set IFS to include that delimiter.)

The shell initially sets IFS to include the blank, tab, and newline characters.

MAIL — The pathname of a file where your mail is deposited. If MAIL is set, then the shell checks to see if anything has been added to the file it names and announces the arrival of new mail each time you return to command level (e.g., by leaving the editor). MAIL is not set automatically; if desired, it should be set (and optionally "exported") in the user's *.profile*. (The **export** command and *.profile* file are discussed later in this chapter.) (The presence of mail in the standard mail file is also announced at login, regardless of whether MAIL is set.)

MAILCHECK — This parameter specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the MAILPATH or MAIL parameters. The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each prompt.

MAILPATH — A colon (:) separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by % and a message that will be printed when the modification time changes. The default message is *you have mail.*

SHACCT — If this parameter is set to the name of a file writable by the user, the shell will write an accounting record in the file for each shell procedure executed. Accounting routines such as **acctcom**(ADM) and **accton**(ADM) can be used to analyze the data collected.

SHELL — When the shell is invoked, it scans the environment for this name. If it is found and there is an 'r' in the file name part of its value, the shell becomes a restricted shell.

PATH — The variable that specifies the search path used by the shell in finding commands. Its value is an ordered list of directory pathnames separated by colons. The shell initializes PATH to the list *:/bin:/usr/bin* where a null argument appears in front of the first colon. A null anywhere in the

path list represents the current directory. On some systems, a search of the current directory is *not* the default and the PATH variable is initialized instead to */bin:/usr/bin.* If you wish to search your current directory last, rather than first, use:

```
PATH=/bin:/usr/bin:
```

Below, the two colons together represent a colon followed by a null, followed by a colon, thus naming the current directory. You could possess a personal directory of commands (say, *$HOME/bin*) and cause it to be searched *before* the other three directories by using:

```
PATH=$HOME/bin::/bin:/usr/bin
```

PATH is normally set in your *.profile* file.

CDPATH — This variable defines the search path for the directory containing **arg**. Alternative directory names are separated by a colon (:). The default path is <null> (specifying the current directory). The current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If **arg** begins with a / then the search path is not used. Otherwise, each directory in the path is searched for **arg**.

PS1 — The variable that specifies what string is to be used as the primary *prompt* string. If the shell is interactive, it prompts with the value of PS1 when it expects input. The default value of PS1 is "$ " (a dollar sign ($) followed by a blank).

PS2 — The variable that specifies the secondary prompt string. If the shell expects more input when it encounters a newline in its input, it prompts with the value of PS2. The default value for this variable is "> " (a greater-than symbol followed by a space).

In general, you should be sure to **export** all of the above variables so that their values are passed to all shells created from your login. Use **export** at the end of your *.profile* file. An example of an **export** statement follows:

```
export HOME IFS MAIL PATH PS1 PS2
```

Predefined Special Variables

Several variables have special meanings; the following are set *only* by the shell:

$# Records the number of arguments passed to the shell, not counting the name of the shell procedure itself. For instance, $# yields the number of the highest set positional parameter. Thus:

```
sh cmd a b c
```

automatically sets $# to 3. One of its primary uses is in checking for the presence of the required number of arguments:

```
if test $# -lt 2
then
      echo 'two or more args required'; exit
fi
```

$? Contains the exit status of the last command executed (also referred to as "return code", "exit code", or "value"). Its value is a decimal string. Most XENIX commands return zero to indicate successful completion. The shell itself returns the current value of $? as its exit status.

$$ The process number of the current process. Because process numbers are unique among all existing processes, this string is often used to generate unique names for temporary files. The operating system provides no mechanism for the automatic creation and deletion of temporary files; a file exists until it is explicitly removed. Temporary files are generally undesirable objects; the XENIX pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur.

The following example illustrates the recommended practice of creating temporary files; note that the directories */usr* and */usr/tmp* are cleared out if the system is rebooted.

```
#     use current process id
#     to form unique temp file
temp=/usr/tmp/$$
ls > $temp
#     commands here, some of which use $temp
rm -f $temp
#     clean up at end
```

$! The process number of the last process run in the background (using the ampersand (&)). This is a string containing from one to five digits.

$- A string consisting of names of execution flags currently turned on in the shell. For example, $- might have the value "xv" if you are tracing your output.

The Shell State

The state of a given instance of the shell includes the values of positional parameters, user-defined variables, environment variables, modes of execution, and the current working directory.

The state of a shell may be altered in various ways. These include changing the working directory with the **cd** command, setting several flags, and by reading commands from the special file, *.profile*, in your login directory.

Changing Directories

The **cd** command changes the current directory to the one specified as its argument. This can and should be used to change to a convenient place in the directory structure. Note that **cd** is often placed within parentheses to cause a subshell to change to a different directory and execute some commands without affecting the original shell.

For example, the first sequence below copies the file */etc/passwd* to */usr/you/passwd*; the second example first changes directory to */etc* and then copies the file:

```
cp /etc/passwd /usr/you/passwd
(cd /etc; cp passwd /usr/you/passwd)
```

Note the use of parentheses. Both command lines have the same effect.

If the shell is reading its commands from a terminal, and the specified directory does not exist (or some component cannot be searched), spelling correction is applied to each component of *directory*, in a search for the "correct" name. The shell then asks whether or not to try and change directory to the corrected directory name; an answer of *n* means "no", and anything else is taken as "yes."

The .profile File

The file named *.profile* is read each time you log in. It is normally used to execute special one-time-only commands and to set and export variables to all later shells. Only after commands are read and executed from *.profile*, does the shell read commands from the standard input—usually the terminal.

If you wish to reset the environment after making a change to the *.profile* file, enter

```
.profile
```

This command eliminates the need to log out and then log in again to execute *.profile*.

Execution Flags

The **set** command lets you alter the behavior of the shell by setting certain shell flags. In particular, the **-x** and **-v** flags may be useful when invoking the shell as a command from the terminal. The flags -x and -v may be **set** by entering:

```
set -xv
```

The same flags may be turned *off* by entering:

```
set +xv
```

These two flags have the following meaning:

-v Input lines are printed as they are read by the shell. This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.

-x Commands and their arguments are printed as they are executed. (Shell control commands, such as **for**, **while**, etc., are not printed, however.) Note that **-x** causes a trace of only those commands that are actually executed, whereas **-v** prints each line of input until a syntax error is detected.

The **set** command is also used to set these and other flags within shell procedures.

A Command's Environment

All variables and their associated values that are known to a command at the beginning of its execution make up its *environment.* This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a shell passes to its child processes are those that have been named as arguments to the **export** command. The **export** command places the named variables in the environments of both the shell *and* all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line. Such variables are placed in the environment of the procedure being invoked. For example:

```
#     keycommand
echo $a $b
```

This is a simple procedure that echoes the values of two variables. If it is invoked as:

```
a=key1 b=key2 keycommand
```

then the resulting output is:

```
key1 key2
```

Keyword parameters are *not* counted as arguments to the procedure and do not affect $#.

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are not reflected in the environment; they are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure. To obtain a list of variables that have been made exportable from the current shell, enter:

```
export
```

You will also get a list of variables that have been made **readonly**. To get a list of name-value pairs in the current environment, enter either:

```
printenv
```

or

```
env
```

Invoking the Shell

The shell is a command and may be invoked in the same way as any other command:

sh *proc* [*arg* ...]	A new instance of the shell is explicitly invoked to read *proc*. Arguments, if any, can be manipulated.
sh -v *proc* [*arg* ...]	This is equivalent to putting "set -v" at the beginning of *proc*. It can be used in the same way for the **-x**, **-e**, **-u**, and **-n** flags.
proc [*arg* ...]	If *proc* is an executable file, and is not a compiled executable program, the effect is similar to that of: sh proc args An advantage of this form is that variables that have been exported in the shell will still be exported from *proc* when this form is used (because the shell only forks to read commands from *proc*). Thus any changes made within *proc* to the values of exported variables will be passed on to subsequent commands invoked from *proc*.

Passing Arguments to Shell Procedures

When a command line is scanned, any character sequence of the form $\$n$ is replaced by the nth argument to the shell, counting the name of the shell procedure itself as $0. This notation permits direct reference to the procedure name and to as many as nine positional parameters. Additional arguments can be processed using the **shift** command or by using a **for** loop.

The **shift** command shifts arguments to the left; i.e., the value of $1 is thrown away, $2 replaces $1, $3 replaces $2, and so on. The highest-numbered positional parameter becomes *unset* ($0 is never shifted). For example, in the shell procedure *ripple* below, **echo** writes its arguments to the standard output.

```
#     ripple command
while test $# != 0
do
      echo $1 $2 $3 $4 $5 $6 $7 $8 $9
      shift
done
```


Lines that begin with a number sign (#) are comments. The looping command, **while**, is discussed in "Conditional Looping: while and until" in this chapter. If the procedure were invoked with:

```
ripple a b c
```

it would print:

```
a b c
b c
c
```

The special shell variable "star" ($*) causes substitution of all positional parameters except $0. Thus, the **echo** line in the *ripple* example above could be written more compactly as:

```
echo $*
```

These two **echo** commands are *not* equivalent: the first prints at most nine positional parameters; the second prints *all* of the current positional parameters. The shell star variable ($*) is more concise and less error-prone. One obvious application is in passing an arbitrary number of arguments to a command. For example:

```
wc $*
```

counts the words of each of the files named on the command line.

It is important to understand the sequence of actions used by the shell in scanning command lines and substituting arguments. The shell first reads input up to a newline or semicolon, and then parses that much of the input. Variables are replaced by their values and then command substitution (via back quotation marks) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next, the shell scans the resulting command line for *internal field separators*, that is, for any characters specified by IFS to break the command line into distinct arguments; *explicit* null arguments (specified by "" or ´´) are retained, while implicit null arguments resulting from evaluation of variables that are null or not set are removed. Then filename generation occurs with all metacharacters being expanded. The resulting command line is then executed by the shell.

Sometimes, command lines are built inside a shell procedure. In this case, it is sometimes useful to have the shell rescan the command line after all the initial substitutions and expansions have been performed. The special command **eval** is available for this purpose. **eval** takes a command line as its argument and simply rescans the line, performing any variable or command substitutions that are specified. Consider the following (simplified) situation:

```
command=who
output=´ | wc -l´
eval $command $output
```

This segment of code results in the execution of the command line:

```
who | wc -l
```

Uses of **eval** can be nested so that a command line can be evaluated several times.

Controlling the Flow of Control

The shell provides several commands that implement a variety of control structures useful in controlling the flow of control in shell procedures. Before describing these structures, a few terms need to be defined.

A *simple command* is any single irreducible command specified by the name of an executable file. I/O redirection arguments can appear in a simple command line and are passed to the shell, *not* to the command.

A *command* is a simple command or any of the shell control commands described below. A *pipeline* is a sequence of one or more commands separated by vertical bars (|). In a pipeline, the standard output of each command but the last is connected (by a *pipe*) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. The exit status of a pipeline is the exit status of last process in the pipeline.

A *command list* is a sequence of one or more pipelines separated by a semicolon (;), an ampersand (&), an "and-if" symbol (&&), or an "or-if" (||) symbol, and optionally terminated by a semicolon or an ampersand. A semicolon causes sequential execution of the previous pipeline. This means that the shell waits for the pipeline to finish before reading the next pipeline. On the other hand, the ampersand (&) causes asynchronous background execution of the preceding pipeline. Thus, both sequential and background execution are allowed. A background pipeline continues execution until it terminates voluntarily, or until its processes are killed.

Other uses of the ampersand include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you enter:

```
nohup cc prog.c&
```

You may continue working while the C compiler runs in the background. A command line ending with an ampersand is immune to interrupts or quits that you might generate by typing **INTERRUPT** or **QUIT**. However, ⟨Ctrl⟩d *will* abort the command if you are operating over a dial-up line or have *stty hupcl*. In this case, it is wise to make the command immune to hang-ups (i.e., logouts) as well. The **nohup** command is used for this purpose. In the above example without **nohup**, if you log out from a dial-up line while **cc** is still executing, **cc** will be killed and your output will disappear.

The ampersand operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of background processes without a compelling reason for doing so.

The and-if and or-if (&& and ||) operators cause conditional execution of pipelines. Both of these are of equal precedence when evaluating command lines (but both are lower than the ampersand (&) and the vertical bar (|)). In the command line:

```
cmd1 || cmd2
```

the first command, *cmd1*, is executed and its exit status examined. Only if *cmd1* fails (i.e., has a nonzero exit status) is *cmd2* executed. Thus, this is a more terse notation for:

```
if    cmd1
      test $? != 0
then
      cmd2
fi
```

The and-if operator (&&) yields a complementary test. For example, in the following command line:

```
cmd1 && cmd2
```

the second command is executed only if the first *succeeds* (and has a zero exit status). In the sequence below, each command is executed in order until one fails:

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line formats and prints two separate documents:

```
{ nroff -mm text1; nroff -mm text2; } | lpr
```

Note that a space is needed after the left brace and that a semicolon should appear before the right brace.

Using the if Statement

The shell provides structured conditional capability with the **if** command. The simplest **if** command has the following form:

if *command-list*
then *command-list*
fi

The command list following the **if** is executed and if the last command in the list has a zero exit status, then the command list that follows **then** is executed. The word **fi** indicates the end of the **if** command.

To cause an alternative set of commands to be executed when there is a nonzero exit status, an **else** clause can be given with the following structure:

if *command-list*
then *command-list*
else *command-list*
fi

Multiple tests can be achieved in an **if** command by using the **elif** clause, although the **case** statement may be better for large numbers of tests. For example:

```
if    test -f "$1"
#                 is $1 a file?
then  pr $1
elif  test -d "$1"
#                 else, is $1 a directory?
then  (cd $1; pr *)
else  echo $1 is neither a file nor a directory
fi
```

The above example is executed as follows: if the value of the first positional parameter is a filename (-f), then print that file; if not, then check to see if it is the name of a directory (-d). If so, change to that directory (cd) and print all the files there (pr *). Otherwise, **echo** the error message.

The **if** command may be nested (but be sure to end each one with a **fi**). The newlines in the above examples of **if** may be replaced by semicolons.

The exit status of the **if** command is the exit status of the last command executed in any **then** clause or **else** clause. If no such command was executed, **if** returns a zero exit status.

Note that an alternate notation for the **test** command uses brackets to enclose the expression being tested. For example, the previous example might have been written as follows:

```
if    [ -f "$1" ]
#                 is $1 a file?
then  pr $1
elif  [ -d "$1" ]
#                 else, is $1 a directory?
then  (cd $1; pr *)
else  echo $1 is neither a file nor a directory
fi
```

Note that a space after the left bracket and one before the right bracket are essential in this form of the syntax.

Using the case Statement

A multiple test conditional is provided by the **case** command. The basic format of the **case** statement is:

case *string* **in**
    pattern) *command-list* ;;
    ...
    pattern) *command-list* ;;
esac

The shell tries to match *string* against each pattern in turn, using the same pattern-matching conventions as in filename generation. If a match is found, the command list following the matched pattern is executed; the double semicolon (;;) serves as a break out of the **case** and is required after each command list except the last. Note that only one pattern is ever matched, and that matches are attempted in order, so that if a star (*) is the first pattern in a **case**, no other patterns are looked at.

More than one pattern may be associated with a given command list by specifying alternate patterns separated by vertical bars (|).

```
case $i in
      *.c)  cc $i
            ;;
      *.h | *.sh)
            : do nothing
            ;;
      *)    echo "$i of unknown type"
            ;;
esac
```

In the above example, no action is taken for the second set of patterns because the null, colon (:) command is specified. The star (*) is used as a default pattern, because it matches any word.

The exit status of **case** is the exit status of the last command executed in the **case** command. If no commands are executed, then **case** has a zero exit status.

Conditional Looping: while and until

A **while** command has the general form:

while *command-list*
do
    command-list
done

The commands in the first *command-list* are executed, and if the exit status of the last command in that list is zero, then the commands in the second *command-list* are executed. This sequence is repeated as long as the exit status of the first *command-list* is zero. A loop can be executed as long as the first command-list returns a nonzero exit status by replacing **while** with **until**.

Any newline in the above example may be replaced by a semicolon. The exit status of a **while** (or **until**) command is the exit status of the last command executed in the second *command-list*. If no such command is executed, **while** (or **until**) has a zero exit status.

Looping Over a List: for

Often, one wishes to perform some set of operations for each file in a set of files, or execute some command once for each of several arguments. The **for** command can be used to accomplish this. The **for** command has the format:

for *variable* **in** *word-list*
do
    command-list
done

Here *word-list* is a list of strings separated by blanks. The commands in the *command-list* are executed once for each word in the *word-list*. *Variable* takes on as its value each word from the word list, in turn. The word list is fixed after it is evaluated the first time. For example, the following **for** loop causes each of the C source files *xec.c*, *cmd.c*, and *word.c* in the current directory to be compared with a file of the same name in the directory */usr/src/cmd/sh*:

```
for CFILE in xec cmd word
do    diff $CFILE.c /usr/src/cmd/sh/$CFILE.c
done
```

Note that the first occurrence of CFILE immediately after the word **for** has no preceding dollar sign, since the name of the variable is wanted and not its value.

You can omit the "**in** *word-list*" part of a **for** command; this causes the current set of positional parameters to be used in place of word-list. This is useful when writing a command that performs the same set of commands for each of an unknown number of arguments.

As an example, create a file named *echo2* that contains the following shell script:

```
for word
do echo $word$word
done
```

Give *echo2* execute status:

chmod +x echo2

Now type the following command:

echo2 ma pa bo fi yo no so ta

The output from this command is:

```
mama
papa
bobo
fifi
yoyo
nono
soso
tata
```

Loop Control: break and continue

The **break** command can be used to terminate execution of a **while** or a **for** loop. The **continue** command immediately starts the execution of the next iteration of the loop. These commands are effective only when they appear between **do** and **done**.

The **break** command terminates execution of the smallest (i.e., innermost) enclosing loop, causing execution to resume after the nearest following unmatched **done**. Exit from *n* levels is obtained by **break** *n*.

The **continue** command causes execution to resume at the nearest enclosing **for**, **while**, or **until** statement, i.e., the one that begins the innermost loop containing the **continue**. You can also specify an argument *n* to **continue** and execution will resume at the *n*th enclosing loop:

```
# This procedure is interactive.
# "Break" and "continue" commands are used
# to allow the user to control data entry.
while true #loop forever
do    echo "Please enter data"
      read response
      case "$response" in
      "done")     break
            # no more data
            ;;
      "")   # just a carriage return,
            # keep on going
            continue
            ;;
      *)    # process the data here
            ;;
      esac
done
```


End-of-File and exit

When the shell reaches the end-of-file in a shell procedure, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The top level shell is terminated by typing a ⟨Ctrl⟩d (which logs the user out of the system).

The **exit** command simulates an end-of-file, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated normally by placing "exit 0" at the end of the file.

Command Grouping: Parentheses and Braces

There are two methods for grouping commands in the shell: parentheses and braces. Parentheses cause the shell to create a subshell that reads the enclosed commands. Both the right and left parentheses are recognized

wherever they appear in a command line—they can appear as literal parentheses *only* when enclosed in quotation marks. For example, if you enter:

```
garble(stuff)
```

the shell prints an error message. Quoted lines, such as:

```
garble"("stuff")"
"garble(stuff)"
```

are interpreted correctly. Other quoting mechanisms are discussed in "Quoting Mechanisms" in this chapter.

This capability of creating a subshell by grouping commands is useful when performing operations without affecting the values of variables in the current shell, or when temporarily changing the working directory and executing commands in the new directory without having to return to the current directory.

The current environment is passed to the subshell and variables that are exported in the current shell are also exported in the subshell. Thus:

```
CURRENTDIR=`pwd`; cd /usr/docs/otherdir;
nohup nroff doc.n > doc.out&; cd $CURRENTDIR
```

and

```
(cd /usr/docs/otherdir; nohup nroff doc.n > doc.out&)
```

accomplish the same result: */usr/docs/otherdir/doc.n* is processed by *nroff* and the output is saved in */usr/docs/otherdir/doc.out*. (Note that **nroff** is a text processing command.) However, the second example automatically puts you back in your original working directory. In the second example above, blanks or newlines surrounding the parentheses are allowed but not necessary. When entering a command line at your terminal, the shell will prompt with the value of the shell variable PS2 if an end parenthesis is expected.

Braces ({ and }) may also be used to group commands together. Both the left and the right brace are recognized *only* if they appear as the first (unquoted) word of a command. The opening brace may be followed by a newline (in which case the shell prompts for more input). Unlike parentheses, no subshell is created for braces: the enclosed commands are simply read by the shell. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command.

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

Defining Functions

The shell includes a function definition capability. Functions are like shell scripts or procedures except that they reside in memory and so are executed by the shell process, not by a separate process. The basic form is:

name ( ) {*list*;}

list can include any of the commands previously discussed. Functions can be defined in one section of a shell script to be called as many times as needed, making them easier to write and maintain. Here is an example of a function called "getyn":

```
# Prompt for yes or no answer - returns non-zero for no
getyn( )    {
      while echo "$* (y/n)? \C" >& 2
      do    read yn rest
            case $yn in
            [yY]) return 0                          ;;
            [nN]) return 1                          ;;
            *)    echo "Please answer y or n" >&2    ;;
            esac
      done
}
```


In this example, the function appends a "(y/n)?" to the output and accepts "Y", "y", "n" or "N" as input, returning a 0 or 1. If the input is anything else, the function prompts the user for the correct input. (Echo should never fail, so the while-loop is effectively infinite.)

Functions are used just like other commands; an invocation of *getyn* might be:

```
getyn "Do you wish to continue" || exit
```

However, unlike other commands, the shell positional parameters **$1, $2,** ..., are set to the arguments of the function. Since an exit in a function will terminate the shell procedure, the return command should be used to return a value back to the procedure.

Input/Output Redirection and Control Commands

The shell normally does *not* fork and create a new shell when it recognizes the control commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input to or output from each command. Also, when redirection of input or output is specified explicitly to a control command, a separate process is spawned to execute that command. Thus, when **if**, **while**, **until**, **case**, and **for** are used in a pipeline consisting of more than one command, the shell forks and a subshell runs the control command. This has two implications:

1. Any changes made to variables within the control command are not effective once that control command finishes (this is similar to the effect of using parentheses to group commands).

2. Control commands run slightly slower when redirected, because of the additional overhead of creating a shell for the control command.

Transfer Between Files: The Dot (.) Command

A command line of the form:

```
. proc
```

causes the shell to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the dot command finishes. This is a good way to gather a number of shell variable initializations into one file. A common use of this command is to reinitialize the top level shell by reading the *.profile* file with:

```
. .profile
```

Interrupt Handling: trap

Shell procedures can use the **trap** command to disable a signal (cause it to be ignored), or redefine its action. The form of the **trap** command is:

trap *arg signal-list*

Here *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers as described in **signal** (S) in the *Programmer's Reference*. The most important of these signals follow:

Number	Signal
0	Exit from the shell
1	HANGUP
2	INTERRUPT character (DELETE or RUB OUT)
3	QUIT (⟨Ctrl⟩\)
9	KILL (cannot be caught or ignored)
11	Segmentation violation (cannot be caught or ignored)
15	Software termination signal

The commands in *arg* are scanned at least once, when the shell first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotation marks to surround these commands. The former inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the shell. The following procedure will print the name of the current directory in the user information as to how much of the job was done:


```
trap 'echo Directory was `pwd` when interrupted' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
      cd $i
         # commands to be executed in directory $i here
done
```

Beware that the same procedure with double rather than single quotation marks does something different. The following prints the name of the directory from which the procedure was first executed:

```
trap "echo Directory was `pwd` when interrupted" 2 3 15
```

A signal 11 can never be trapped, because the shell itself needs to catch it to deal with memory allocation. Zero is interpreted by the **trap** command as a signal generated by exiting from a shell. This occurs either with an **exit** command, or by "falling through" to the end of a procedure. If *arg* is not specified, then the action taken upon receipt of any of the signals in the signal list is reset to the default system action. If *arg* is an explicit null string (''or ""), then the signals in the signal list are ignored by the shell.

The **trap** command is most frequently used to make sure that temporary files are removed upon termination of a procedure. The preceding example would be written more typically as follows:

```
temp=$HOME/temp/$$
trap 'rm -f $temp; exit' 0 1 2 3 15
ls > $temp
     # commands that use $temp here
```

In this example, whenever signal 1 (hangup), 2 (interrupt), 3 (quit), or 15(terminate) is received by the shell procedure, or whenever the shell procedure is about to exit, the commands enclosed between the single quotation marks are executed. The **exit** command must be included, or else the shell continues reading commands where it left off when the signal was received.

Sometimes the shell continues reading commands after executing trap commands. The following procedure takes each directory in the current directory, changes to that directory, prompts with its name, and executes commands typed at the terminal until an end-of-file (⟨Ctrl⟩d) or an interrupt is received. An end-of-file causes the **read** command to return a nonzero exit status, and thus the **while** loop terminates and the next directory cycle is initiated. An interrupt is ignored while executing the requested commands, but causes termination of the procedure when it is waiting for input:


```
d=`pwd`
for i in *
do    if test -d $d/$i
      then cd $d/$i
           while echo "$i:"
                 trap exit 2
                 read x
           do    trap : 2
                 # ignore interrupts
                 eval $x
           done
      fi
done
```

Several **traps** may be in effect at the same time: if multiple signals are received simultaneously, they are serviced in numerically ascending order. To determine which traps are currently set, enter:

```
trap
```

It is important to understand some things about the way in which the shell implements the **trap** command. When a signal (other than 11) is received by the shell, it is passed on to whatever child processes are currently executing.

When these (synchronous) processes terminate, normally or abnormally, the shell polls any traps that happen to be set and executes the appropriate **trap** commands. This process is straightforward, except in the case of traps set at the command (outermost, or login) level. In this case, it is possible that no child process is running, so before the shell polls the traps, it waits for the termination of the first process spawned *after* the signal was received.

When a signal is redefined in a shell script, this does not redefine the signal for programs invoked by that script; the signal is merely passed along. A disabled signal is not passed.

For internal commands, the shell normally polls traps on completion of the command. An exception to this rule is made for the **read** command, for which traps are serviced immediately, so that **read** can be interrupted while waiting for input.

Shell Script Example

The following is a good shell script for handling signals.

```
:
#
#             Set signal handlers for shell script
#
trap "echo
trap "echo
trap "echo
#
#       Note:  If you cd to a different directory you may want to
#       reset the trap for SIGQUIT so it will find the "core" file.
#       To do this you would put the same line below the cd command
#       in the shell script.
#
trap "echo

echo " Going into loop0
while true
do
        cd /tmp
        trap "echo
        lf
        cd /usr
        trap "echo
        lf
        sleep 1
done
echo " Leaving the loop 0
exit 0
```

Special Shell Commands

There are several special commands that are *internal* to the shell, some of which have already been mentioned. The shell does not fork to execute these commands, so no additional processes are spawned. These commands should be used whenever possible, because they are, in general, faster and more efficient than other XENIX commands.

Several of the special commands have already been described because they affect the flow of control. They are dot (**.**), **break, continue, exit,** and **trap.** The **set** command is also a special command. Descriptions of the remaining special commands are given here:

: — The null command. This command does nothing and can be used to insert comments in shell procedures. Its exit status is zero (true). Its utility as a comment character has largely been supplanted by the number sign (#) which can be used to insert comments to the end-of-line. Beware: any arguments to the null command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place, just as in other commands.

cd *arg* — Make *arg* the current directory. If *arg* is not a valid directory, or the user is not authorized to access it, a nonzero exit status is returned. Specifying **cd** with no *arg* is equivalent to entering "cd $HOME" which takes you to your home directory.

exec *arg* ... — If *arg* is a command, then the shell executes the command without forking and returning to the current shell. This is effectively a "goto" and no new process is created. Input and output redirection arguments are allowed on the command line. If *only* input and output redirection arguments appear, then the input and output of the shell itself are modified accordingly.

hash [*-r*] *name*

For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The **-r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented. *Hits* is the number of times a command has been invoked by the shell process. *Cost* is a measure of the work required to locate a command in the search path. There are certain situations which require that the stored location of a command be recalculated. Commands for which this will be done are indicated by an asterisk (*) adjacent to the *hits* information. *Cost* will be incremented when the recalculation is done.

newgrp *arg* ...

The **newgrp** command is executed, replacing the shell. **Newgrp** in turn creates a new shell. Beware: only environment variables will be known in the shell created by the **newgrp** command. Any variables that were exported will no longer be marked as such.

pwd

Print the current working directory. See **pwd**(C) for usage and description.

read *var* ...

One line (up to a newline) is read from the standard input and the first word is assigned to the first variable, the second word to the second variable, and so on. All words left over are assigned to the *last* variable. The exit status of **read** is zero unless an end-of-file is read.

readonly *var* ...

The specified variables are made **readonly** so that no subsequent assignments may be made to them. If no arguments are given, a list of all **readonly** and of all exported variables is given.

return *n*

Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.

times

The accumulated user and system times for processes run from the current shell are printed.

type *name*

For each *name*, indicate how it would be interpreted if used as a command name.

ulimit [-f] *n*

This imposes a size limit of *n* blocks on files written. The **-f** flag imposes a size limit of *n* blocks on files written by child processes (files of any size may be read). With no argument, the current limit is printed. If no option is given and a number is specified, **-f** is assumed.

umask *nnn*

The user file creation mask is set to *nnn*. If *nnn* is omitted, then the current value of the mask is printed. This bit-mask is used to set the default permissions when creating files. For example, an octal umask of 137 corresponds to the following bit-mask and permission settings for a newly created file:

	user	group	other
Octal	1	3	7
bit-mask	001	011	111
permissions	rw-	r--	---

See **umask**(C) in the *XENIX Reference* for information on the value of *nnn*.

unset *name*

For each *name*, remove the corresponding variable or function. The variables PATH, PS1, PS2, MAILCHECK and IFS cannot be unset.

wait *n*

The shell waits for all currently active child processes to terminate. If *n* is specified, the shell waits for the specified process to terminate. The exit status of *wait* is always zero if *n* is not given; otherwise it is the exit status of child *n*.

Creation and Organization of Shell Procedures

A shell procedure can be created in two simple steps. The first is building an ordinary text file. The second is changing the *mode* of the file to make it *executable*, thus permitting it to be invoked by:

```
proc args
```

rather than

```
sh proc args
```

The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for frequently-used ones. For example, create a file named *mailall* with the following contents:

```
LETTER=$1
shift
for i in $*
do mail $i < $LETTER
done
```

Next enter:

```
chmod +x mailall
```

The new command might then be invoked from within the current directory by entering:

```
mailall letter joe bob
```

Here *letter* is the name of the file containing the message you want to send, and *joe* and *bob* are people you want to send the message to. Note that shell procedures must always be at least readable, so that the shell itself can read commands from the file.

If *mailall* were thus created in a directory whose name appears in the user's PATH variable, the user could change working directories and still invoke the *mailall* command.

Shell procedures are often used by users running the **csh**. However, if the first character of the procedure is a # (comment character), the **sh** assumes the procedure is a **csh** script, and invokes */bin/csh* to execute it. Always start **sh** procedures with some other character if **csh** users are to run the procedure at any time. This invokes the standard shell */bin/sh*.

Shell procedures may be created dynamically. A procedure may generate a file of commands, invoke another instance of the shell to execute that file, and then remove it. An alternate approach is that of using the *dot* command (.) to make the current shell read commands from the new file, allowing use of existing shell variables and avoiding the spawning of an additional process for another shell.

Many users prefer writing shell procedures to writing programs in C or other traditional languages. This is true for several reasons:

1. A shell procedure is easy to create and maintain because it is only a file of ordinary text.

2. A shell procedure has no corresponding object program that must be generated and maintained.

3. A shell procedure is easy to create quickly, use a few times, and then remove.

4. Because shell procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and shell procedures are named *bin*. This name is derived from the word "binary", and is used because compiled and executable programs are often called "binaries" to distinguish them from program source files. Most groups of users sharing common interests have one or more *bin* directories set up to hold common procedures. Some users have their PATH variable list several such directories. Although you can have a number of such directories, it is unwise to go overboard: it may become difficult to keep track of your environment and efficiency may suffer.

More About Execution Flags

There are several execution flags available in the shell that can be useful in shell procedures:

-e This flag causes the shell to exit immediately if any command that it executes exits with a nonzero exit status. This flag is useful for shell procedures composed of simple command lines; it is not intended for use in conjunction with other conditional constructs.

-u This flag causes unset variables to be considered errors when substituting variable values. This flag can be used to effect a global check on variables, rather than using conditional substitution to check each variable.

-t This flag causes the shell to exit after reading and executing the commands on the remainder of the current input line. This flag is typically used by C programs which call the shell to execute a single command.

-n This is a "don't execute" flag. On occasion, one may want to check a procedure for syntax errors, but not execute the commands in the procedure. Using "set -nv" at the beginning of a file will accomplish this.

-k This flag causes all arguments of the form *variable=value* to be treated as keyword parameters. When this flag is *not* set, only such arguments that appear before the command name are treated as keyword parameters.

Supporting Commands and Features

Shell procedures can make use of any XENIX command. The commands described in this section are either used especially frequently in shell procedures, or are explicitly designed for such use.

Conditional Evaluation: test

The **test** command evaluates the expression specified by its arguments and, if the expression is true, **test** returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. **test** also returns a nonzero exit status if it has no arguments. Often it is convenient to use the **test** command as the first command in the command list following an **if** or a **while**. Shell variables used in **test** expressions should be enclosed in double quotation marks if there is any chance of their being null or not set.

The square brackets may be used as an alias to **test**, so that:

[*expression*]

has the same effect as:

test *expression*

Note that the spaces before and after the *expression* in brackets are essential.

The following is a partial list of the options that can be used to construct a conditional expression:

-r *file*	True if the named file exists and is readable by the user.
-w *file*	True if the named file exists and is writable by the user.
-x *file*	True if the named file exists and is executable by the user.
-s *file*	True if the named file exists and has a size greater than zero.

-d *file*	True if the named file is a directory.
-f *file*	True if the named file is an ordinary file.
-z *s1*	True if the length of string *s1* is zero.
-n *s1*	True if the length of the string *s1* is nonzero.
-t *fildes*	True if the open file whose file descriptor number is *fildes* is associated with a terminal device. If *fildes* is not specified, file descriptor 1 is used by default.
s1 = *s2*	True if strings *s1* and *s2* are identical.
s1 != *s2*	True if strings *s1* and *s2* are *not* identical.
s1	True if *s1* is *not* the null string.
n1 **-eq** *n2*	True if the integers *n1* and *n2* are algebraically equal; other algebraic comparisons are indicated by **-ne** (not equal), **-gt** (greater than), **-ge** (greater than or equal to), **-lt** (less than), and **-le** (less than or equal to).

These may be combined with the following operators:

!	Unary negation operator.
-a	Binary logical AND operator.
-o	Binary logical OR operator; it has lower precedence than the logical AND operator (-a).
(*expr*)	Parentheses for grouping; they must be escaped to remove their significance to the shell. In the absence of parentheses, evaluation proceeds from left to right.

Note that all options, operators, filenames, etc. are separate arguments to **test**.

Echoing Arguments

The **echo** command has the following syntax:

echo [*options*] [*args*]

echo copies its arguments to the standard output, each followed by a single space, except for the last argument, which is normally followed by a newline. You can use it to prompt the user for input, to issue diagnostics in shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic.

You can replace the **ls** command with

```
echo *
```

because the latter is faster and prints fewer lines of output.

The **-n** option to **echo** removes the newline from the end of the echoed line. Thus, the following two commands prompt for input and then allow entering on the same line as the prompt:

```
echo -n 'enter name:'
read name
```

The **echo** command also recognizes several escape sequences described in **echo** (C) in the *XENIX Reference*.

Expression Evaluation: expr

The **expr** command provides arithmetic and logical operations on integers and some pattern-matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output; **expr** can be used inside grave accents to set a variable. Some typical examples follow:

```
#     increment $A
A=`expr $a + 1`
#     put third through last characters of
#     $1 into substring
substring=`expr "$1" : '..\(.*\)' `
#     obtain length of $1
c=`expr "$1" :  '.* ' `
```

The most common uses of **expr** are in counting iterations of a loop and in using its pattern-matching capability to pick apart strings.

True and False

The **true** and **false** commands perform the functions of exiting with zero and nonzero exit status, respectively. The **true** and **false** commands are often used to implement unconditional loops. For example, you might enter:

```
while true
do echo forever
done
```

This will echo "forever" on the screen until an INTERRUPT is entered.

In-Line Input Documents

Upon seeing a command line of the form:

command << *eofstring*

where *eofstring* is any arbitrary string, the shell will take the subsequent lines as the standard input of *command* until a line is read consisting only of *eofstring*. (By appending a minus (-) to the input redirection symbol (<<), leading tabs are deleted from each line of the input document before the shell passes the line to *command*.)

The shell creates a temporary file containing the input document and performs variable and command substitution on its contents before passing it to the command. Pattern matching on filenames is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, you may quote any character of *eofstring*:

```
command <<\eofstring
```

The in-line input document feature is especially useful for small amounts of input data, where it is more convenient to place the data in the shell procedure than to keep it in a separate file. For instance, you could enter:

```
cat <<- xx
        This message will be printed on the
        terminal with leading tabs removed.
xx
```

This in-line input document feature is most useful in shell procedures. Note that in-line input documents may not appear within grave accents.

Input/Output Redirection Using File Descriptors

We mentioned above that a command occasionally directs output to some file associated with a file descriptor other than 1 or 2. In languages such as C, one can associate output with any file descriptor by using the **write** (S) system call (see the *Programmer's Reference*). The shell provides its own mechanism for creating an output file associated with a particular file descriptor. By entering:

fd1 >& *fd2*

where *fd1* and *fd2* are valid file descriptors, one can direct output that would normally be associated with file descriptor *fd1* to the file associated with *fd2*. The default value for *fd1* and *fd2* is 1. If, at run time, no file is associated with *fd2*, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by entering:

```
command 2>&1
```

If you wanted to redirect both standard output and standard error output to the same file, you would enter:

```
command 1>file 2>&1
```

The order here is significant: first, file descriptor 1 is associated with *file*; then file descriptor 2 is associated with the same file as is currently associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file*, because at the time of the error output redirection, file descriptor 1 still would have been associated with the terminal.

This mechanism can also be generalized to the redirection of standard input. You could enter:

fda <& fdb

to cause both file descriptors *fda* and *fdb* to be associated with the same input file. If *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for a command that uses two or more input sources.

Conditional Substitution

Normally, the shell replaces occurrences of $*variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution, dependent upon whether the variable is set or not null. By definition, a variable is set if it has ever been assigned a value. The value of a variable can be the null string, which may be assigned to a variable in anyone of the following ways:

```
A=
bcd=""
efg=''
set '' ""
```


The first three examples assign null to each of the corresponding shell variables. The last example sets the first and second positional parameters to null. The following conditional expressions depend upon whether a variable is set and not null. Note that the meaning of braces in these expressions differs from their meaning when used in grouping shell commands. *Parameter* as used below refers to either a digit or a variable name.

${*variable*:-*string*}	If *variable* is set and is nonnull, then substitute the value $*variable* in place of this expression. Otherwise, replace the expression with *string*. Note that the value of *variable* is *not* changed by the evaluation of this expression.
${*variable*:=*string*}	If *variable* is set and is nonnull, then substitute the value $*variable* in place of this expression. Otherwise, set *variable* to *string*, and then substitute the value $*variable* in place of this expres-

sion. Positional parameters may not be assigned values in this fashion.

${*variable*:?*string*} If *variable* is set and is nonnull, then substitute the value of *variable* for the expression. Otherwise, print a message of the form

variable: *string*

and exit from the current shell. (If the shell is the login shell, it is not exited.) If *string* is omitted in this form, then the message

variable: parameter null or not set

is printed instead.

${*variable*:+*string*} If *variable* is set and is nonnull, then substitute *string* for this expression. Otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon. In this variation, the shell does not check whether the variable is null or not; it only checks whether the variable has ever been set.

The two examples below illustrate the use of this facility:

1. This example performs an explicit assignment to the PATH variable:

   PATH=${PATH:-':/bin:/usr/bin'}

   This says, if PATH has ever been set and is not null, then it keeps its current value; otherwise, set it to the string ":/bin:/usr/bin".

2. This example automatically assigns the HOME variable a value:

   cd ${HOME:='/usr/gas'}

   If HOME is set, and is not null, then change directory to it. Otherwise set HOME to the given value and change directory to it.

Invocation Flags

There are five flags that may be specified on the command line when invoking the shell. These flags may not be turned on with the **set** command:

-i If this flag is specified, or if the shell's input and output are both attached to a terminal, the shell is *interactive*. In such a shell, INTERRUPT (signal 2) is caught and ignored, and TERMINATE (signal 15) and QUIT (signal 3) are ignored.

-s If this flag is specified or if no input/output redirection arguments are given, the shell reads commands from standard input. Shell output is written to file descriptor 2. All remaining arguments specify the positional parameters.

-c When this flag is turned on, the shell reads commands from the first string following the flag. Remaining arguments are ignored.

-t When this flag is on, a single command is read and executed, then the shell exits. This flag is not useful interactively, but is intended for use with C programs.

-r If this flag is present the shell is a restricted shell (see **rsh** (C)).

Effective and Efficient Shell Programming

This section outlines strategies for writing efficient shell procedures, ones that do not waste resources in accomplishing their purposes. The primary reason for choosing a shell procedure to perform a specific function is to achieve a desired result at a minimum human cost. Emphasis should always be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, you should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume an inordinate amount of a system's resources.

The same kind of iteration cycle should be applied to shell procedures as to other programs: write code, measure it, and optimize only the *few* important parts. The user should become familiar with the **time** command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by variations in system load.

Number of Processes Generated

When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping, and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is the alphabetical list of those that are built in:

break	case	cd	continue	echo
eval	exec	exit	export	for
if	read	readonly	return	set
shift	test	times	trap	umask
until	wait	while	.	:
{ }				

Parentheses, ( ), are built into the shell, but commands enclosed within them are executed as a child process, i.e., the shell does a **fork**, but no **exec**. Any command not in the above list requires both **fork** and **exec**.

The user should always have at least a vague idea of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes created (not necessarily simultaneously) can be described by:

processes = (k*n) + c

where *k* and *c* are constants, and *n* may be the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k*, sometimes to zero.

Any procedure whose complexity measure includes n^{-2} terms or higher powers of *n* is likely to be intolerably expensive.

As an example, here is an analysis of a procedure named *split*, whose text is given below:

```
:
#     split
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$
            # read stdin into temp file
            # save original lengths of $1, $2
if test -s "$1"
then start1=`wc -l < $1`
fi
if test -s "$2"
then start2=`wc -l < $2`
fi
grep "$b" temp$$ >> $1
            # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
            # lines without letters onto $2
total=" `wc-l< temp$$` "
end1=" `wc-l<$1` "
end2=" `wc-l<$2` "
lost=" `expr $total - \($end1 - $start1\) \
- \($end2 - $start2\)` "
echo "$total read, $lost thrown away"
```


For each iteration of the loop, there is one **expr** plus either an **echo** or another **expr**. One additional **echo** is executed at the end. If *n* is the number of lines of input, the number of processes is 2*n + 1.

Some types of procedures should *not* be written using the shell. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C. Shell procedures should not be used to scan or build files a character at a time.

Number of Data Bytes Accessed

It is worthwhile to consider any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes, rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the *shrinkers* first when

the order is irrelevant. For instance, the second of the following examples is likely to be faster because the input to **sort** will be much smaller:

```
sort file | grep pattern
grep pattern file | sort
```

Shortening Directory Searches

Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of **cd,** the *change directory* command, can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands:

```
ls -l /usr/bin/* >/dev/null
cd /usr/bin; ls -l * >/dev/null
```

The second command will run faster because of the fewer directory searches.

Directory-Search Order and the PATH Variable

The PATH variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result may be a great increase in system overhead.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current PATH variable. As an example, consider the effect of invoking **nroff** (i.e., */usr/bin/nroff*) when the value of PATH is ":/bin:/usr/bin". The sequence of directories read is:

```
.
/
/bin
/
/usr
/usr/bin
```

This is a total of six directories. A long path list assigned to PATH can increase this number significantly.

The vast majority of command executions are of commands found in */bin* and, to a somewhat lesser extent, in */usr/bin*. Careless PATH setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best with respect to the efficiency of command searches:

```
:/usr/john/bin:/usr/localbin:/bin:/usr/bin
:/bin:/usr/john/bin:/usr/localbin:/usr/bin
:/bin:/usr/bin:/usr/john/bin:/usr/localbin
/bin::/usr/bin:/usr/john/bin:/usr/localbin
```

The first one above should be avoided. The others are acceptable and the choice among them is dictated by the rate of change in the set of commands kept in */bin* and */usr/bin*.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by setting the PATH variable inside the procedure so that the fewest possible directories are searched in an optimum order.

Good Ways to Set Up Directories

It is wise to avoid directories that are larger than necessary. You should be aware of several special sizes. A directory that contains entries for up to 30 files (plus the required . and ..) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a small directory; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink. This is very important to understand, because if your directory ever exceeds either the 30 or 286 thresholds, searches will be inefficient; furthermore, even if you delete files so that the number of files is less than either threshold, the system will still continue to treat the directory inefficiently.

Shell Procedure Examples

The power of the XENIX shell command language is most readily seen by examining how many labor-saving XENIX utilities can be combined to perform powerful and useful commands with very little programming effort. This section gives examples of procedures that do just that. By studying these examples, you will gain insight into the techniques and shortcuts that can be used in programming shell procedures (also called "scripts"). Note the use of the null command (:) to begin each shell procedure and the use of the number sign (#) to introduce comments.

It is intended that the following steps be carried out for each procedure:

1. Place the procedure in a file with the indicated name.
2. Give the file execute permission with the **chmod** command.
3. Move the file to a directory in which commands are kept, such as your own *bin* directory.
4. Make sure that the path of the *bin* directory is specified in the PATH variable found in *.profile*.
5. Execute the named command.

BINUNIQ

```
:
ls /bin /usr/bin | sort | uniq -d
```

This procedure determines which files are in both */bin* and */usr/bin*. It is done because files in */bin* will "override" those in */usr/bin* during most searches and duplicates need to be weeded out. If the */usr/bin* file is obsolete, then space is being wasted; if the */bin* file is outdated by a corresponding entry in */usr/bin* then the wrong version is being run and, again, space is being wasted. This is also a good demonstration of "sort | uniq" to find matches and duplications.

COPYPAIRS

```
:
#     Usage: copypairs file1 file2 ...
#     Copies file1 to file2, file3 to file4, ...
while test "$2" != ""
do
      cp $1 $2
      shift; shift
done
if test "$1" != ""
      then echo "$0: odd number of arguments" >&2
fi
```

This procedure illustrates the use of a **while** loop to process a list of positional parameters that are somehow related to one another. Here a **while** loop is much better than a **for** loop, because you can adjust the positional parameters with the **shift** command to handle related arguments.

COPYTO


```
:
#     Usage: copyto dir file ...
#     Copies argument files to "dir",
#     making sure that at least
#     two arguments exist, that "dir" is a directory,
#     and that each additional argument
#     is a readable file.
if test $# -lt 2
      then  echo "$0: usage: copyto directory file ...">&2
elif test ! -d $1
      then  echo "$0: $1 is not a directory";>&2
else  dir=$1; shift
      for eachfile
      do    cp $eachfile $dir
      done
fi
```

This procedure uses an **if** command with several parts to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to **copyto** but the first; the original **$1** is shifted off.

DISTINCT1

```
:
#     Usage: distinct1
#     Reads standard input and reports list of
#     alphanumeric strings that differ only in case,
#     giving lowercase form of each.
tr -cs 'A-Za-z0-9' '\012' | sort -u | \
tr 'A-Z' 'a-z' | sort | uniq -d
```

This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. Note the use of the backslash at the end of the first line as the line continuation character. It may not be immediately obvious how this command works. You may wish to consult **tr**(C), **sort**(C), and **uniq**(C) in the *XENIX Reference* if you are completely unfamiliar with these commands. The **tr** command translates all characters except letters and digits into newline characters, and then squeezes out repeated newline characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The "uniq -d" prints (once) only those lines that occur more than once, yielding the desired list.

The process of building such a pipeline relies on the fact that pipes and files can usually be interchanged. The first line below is equivalent to the last two lines, assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3

cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3
rm temp[123]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output.

Although pipelines can give a concise notation for complex processes, you should exercise some restraint, since such practice often yields incomprehensible code.

DRAFT

```
:
#     Usage: draft file(s)
#     Print manual pages for Diablo printer.
for i in $*
      do nroff -man $i | lpr
done
```

Users often write this kind of procedure for convenience in dealing with commands that require the use of distinct flags that cannot be given default values that are reasonable for all (or even most) users.

EDFIND

```
:
#     Usage: edfind file arg
#     Finds the last occurrence in "file" of a line
#     whose beginning matches "arg", then prints
#     3 lines (the one before, the line itself,
#     and the one after)
ed - $1 << -EOF
      ?^$2?
      -,+p
      q
EOF
```

This illustrates the practice of using **ed** in-line input scripts into which the shell can substitute the values of variables.

EDLAST

```
:
#     Usage: edlast file
#     Prints the last line of file,
#     then deletes that line.
ed - $1 <<-\!
      $p
      $d
      w
      q
!
echo done
```

This procedure illustrates taking input from within the file itself up to the exclamation point (!). Variable substitution is prohibited within the input text because of the backslash.

FSPLIT


```
:
#     Usage: fsplit file1 file2
#     Reads standard input and divides it into 3 parts
#     by appending any line containing at least one letter
#     to file1, appending any line containing digits but
#     no letters to file2, and by throwing the rest away.
count=0 gone=0
while read next
do
      count="`expr $count + 1`"
      case "$next"  in
      *[A-Za-z]*)
            echo  "$next"  >> $1 ;;
      *[0-9]*)
            echo  "$next"  >> $2 ;;
      *)
            gone="`expr $gone + 1`"
      esac
done
echo "$count lines read, $gone thrown away"
```

Each iteration of the loop reads a line from the input and analyzes it. The loop terminates only when **read** encounters an end-of-file. Note the use of the **expr** command.

Do not use the shell to read a line at a time unless you must because it can be an extremely slow process.

LISTFIELDS

```
:
grep $* | tr ":" "\012"
```

This procedure lists lines containing any desired entry that is given to it as an argument. It places any field that begins with a colon on a newline. Thus, if given the following input:

joe newman: 13509 NE 78th St: Redmond, Wa 98062

listfields will produce this:

joe newman
13509 NE 78th St
Redmond, Wa 98062

Note the use of the **tr** command to transpose colons to linefeeds.

MKFILES

```
:
#     Usage: mkfiles pref [quantity]
#     Makes "quantity" files, named pref1, pref2, ...
#     Default is 5 as determined on following line.
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
      > $1$i
      i="`expr $i + 1`"
done
```

The *mkfiles* procedure uses output redirection to create zero-length files. The **expr** command is used for counting iterations of the **while** loop.

NULL

```
:
#     Usage: null files
#     Create each of the named files as an empty file.
for eachfile
do
      >$eachfile
done
```

This procedure uses the fact that output redirection creates the (empty) output file if a file does not already exist.

PHONE

```
:
#     Usage: phone initials ...'
#     Prints the phone numbers of the
#     people with the given initials.
echo 'initsext   home'
grep "$1" <<END
      jfk   1234  999-2345
      lbj   2234  583-2245
      hst   3342  988-1010
      jqa   4567  555-1234
END
```

This procedure is an example of using an in-line input script to maintain a small database.

TEXTFILE

```
:
if test "$1" = "-s"
then
#     Return condition code
      shift
      if test -z "`$0 $*` " # check return value
      then
            exit 1
      else
            exit 0
      fi
fi

if test $# -lt 1
then  echo "$0: Usage: $0 [ -s ] file ..." 1>&2
      exit 0
fi

file $* | fgrep ' text' | sed 's/: .*//'
```


To determine which files in a directory contain only textual information, *textfile* filters argument lists to other commands. For example, the following command line will print all the text files in the current directory:

```
pr `textfile *` | lpr
```

This procedure also uses an **-s** flag which silently tests whether any of the files in the argument list is a text file.

WRITEMAIL

```
:
#     Usage: writemail message user
#     If user is logged in,
#     writes message to terminal;
#     otherwise, mails it to user.
echo "$1" | { write "$2" || mail "$2" ;}
```

This procedure illustrates the use of command grouping. The message specified by $1 is piped to both the **write** command and, if **write** fails, to the **mail** command.

Shell Grammar

item: *word*
input-output
name = *value*

simple-command: *item*
simple-command item

command: *simple-command*
(*command-list*)
{ *command-list* }
for *name* **do** *command-list* **done**
for *name* **in** *word* **do** *command-list* **done**
while *command-list* **do** *command-list* **done**
until *command-list* **do** *command-list* **done**
case *word* **in** *case-part* **esac**
if *command-list* **then** *command-list else-part* **fi**

pipeline: *command*
pipeline | *command*

andor: *pipeline*
andor && *pipeline*
andor || *pipeline*

command-list: *andor*
command-list ;
command-list &
command-list ; *andor*
command-list & *andor*

input-output: > *file*
< *file*
<< *word*
>> *file*
digit > *file*
digit < *file*
digit >> *file*

file: *word*
& *digit*
& –

case-part: *pattern*) *command-list* ;;

pattern: *word*
pattern | *word*

else-part: **elif** *command-list* **then** *command-list else-part*
else *command-list*
empty

empty:

word: *a sequence of nonblank characters*

name: *a sequence of letters, digits, or underscores starting with a letter*

digit: **0 1 2 3 4 5 6 7 8 9**

5

Metacharacters and Reserved Words

1. Syntactic

|	Pipe symbol
&&	And-if symbol
||	Or-if symbol
;	Command separator
;;	Case delimiter
&	Background commands
( )	Command grouping
<	Input redirection
<<	Input from a here document
>	Output creation
>>	Output append
#	Comment to end of line

2. Patterns

*	Match any character(s) including none
?	Match any single character
[...]	Match any of enclosed characters

3. Substitution

${...}	Substitute shell variable
`...`	Substitute command output

4. Quoting

\	Quote next character as literal with no special meaning
'...'	Quote enclosed characters excepting the back quotation marks (')
"..."	Quote enclosed characters excepting: $`\"

5. Reserved words

if	**esac**
then	**for**
else	**while**
elif	**until**
fi	**do**
case	**done**
in	**{ }**

Chapter 6

The C-Shell

Introduction

The C-shell program, **csh,** is a command language interpreter. The C-shell, like the standard XENIX shell *sh*, is an interface between you and the XENIX commands and programs. It translates command lines entered at a terminal into corresponding system actions, gives you access to information, such as your login name, home directory, and mailbox, and lets you construct shell procedures for automating system tasks.

This appendix explains how to use the C-shell. It also explains the syntax and function of C-shell commands and features, and shows how to use these features to create shell procedures. The C-shell is fully described in **csh** (C) in the *XENIX Reference*.

Invoking the C-shell

You can invoke the C-shell from another shell by using the **csh** command. To invoke the C-shell, enter:

csh

at the standard shell's command line. You can also direct the system to invoke the C-shell for you when you log in. If you have given the C-shell as your login shell in your */etc/passwd* file entry, the system automatically starts the shell when you log in.

After the system starts the C-shell, the shell searches your home directory for the command files *.cshrc* and *.login*. If the shell finds the files, it executes the commands contained in them, then displays the C-shell prompt.

The *.cshrc* file typically contains the commands you wish to execute each time you start a C-shell, and the *.login* file contains the commands you wish to execute after logging in to the system. For example, the following is the contents of a typical *.login* file:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
set time=15
set history=10
mail
```

This file contains several **set** commands. The **set** command is executed directly by the C-shell; there is no corresponding XENIX program for this command. **Set** sets the C-shell variable "ignoreeof" which shields the C-shell from logging out if Ctrl-d is hit. Instead of Ctrl-d, the **logout** command is used to log out of the system. By setting the "mail" variable, the C-shell is notified that it is to watch for incoming mail and notify you if new mail arrives.

Next the C-shell variable "time" is set to 15 causing the C-shell to automatically print out statistics lines for commands that execute for at least 15 seconds of CPU time. The variable "history" is set to 10 indicating that the C-shell will remember the last 10 commands typed in its history list, (described later).

Finally, the XENIX *mail* program is invoked.

When the C-shell finishes processing the *.login* file, it begins reading commands from the terminal, prompting for each with:

```
%
```

When you log out (by giving the **logout** command) the C-shell prints:

```
logout
```

and executes commands from the file *.logout* if it exists in your home directory. After that, the C-shell terminates and logs you off the system.

Using Shell Variables

The C-shell maintains a set of variables. For example, in the above discussion, the variables "history" and "time" had the values 10 and 15. Each C-shell variable has as its value an array of zero or more strings. C-shell variables may be assigned values by the **set** command, which has several forms, the most useful of which is:

set *name* = *value*

C-shell variables may be used to store values that are to be used later in commands through a substitution mechanism. The C-shell variables most commonly referenced are, however, those that the C-shell itself refers to. By changing the values of these variables you can directly affect the behavior of the C-shell.

One of the most important variables is "path". This variable contains a list of directory names. When you enter a command name at your terminal, the C-shell examines each named directory in turn, until it finds an executable file whose name corresponds to the name you entered. The **set** command with no arguments displays the values of all variables currently defined in the C-shell.

The following example file shows typical default values:

```
argv ()
home /usr/bill
path (. /bin /usr/bin)
prompt     %
shell /bin/csh
status     0
```

This output indicates that the variable "path" begins with the current directory indicated by dot (.), then */bin*, and */usr/bin*. Your own local commands may be in the current directory. Normal XENIX commands reside in */bin* and */usr/bin*.

Sometimes a number of locally developed programs reside in the directory */usr/local*. If you want all C-shells that you invoke to have access to these new programs, place the command:

set path=(. /bin /usr/bin /usr/local)

in the *.cshrc* file in your home directory. Try doing this, then logging out and back in. Enter:

set

to see that the value assigned to "path" has changed.

You should be aware that when you log in the C-shell examines each directory that you insert into your path and determines which commands are contained there, except for the current directory which the C-shell treats specially. This means that if commands are added to a directory in your search path after you have started the C-shell, they will not necessarily be found. If you wish to use a command which has been added after you have logged in, you should give the command:

rehash

to the C-shell. **rehash** causes the shell to recompute its internal table of command locations, so that it will find the newly added command. Since the C-shell has to look in the current directory on each command anyway, placing it at the end of the path specification usually works best and reduces overhead.

Other useful built in variables are "home" which shows your home directory, and "ignoreeof" which can be set in your *.login* file to tell the C-shell not to exit when it receives an end-of-file from a terminal. The variable "ignoreeof" is one of several variables whose value the C-shell does not care about; the C-shell is only concerned with whether these variables are set or unset. Thus, to set "ignoreeof" you simply enter:

set ignoreeof

and to unset it enter:

unset ignoreeof

Some other useful built-in C-shell variables are "noclobber" and "mail".

The syntax:

>filename

which redirects the standard output of a command just as in the regular shell, overwrites and destroys the previous contents of the named file. In this way,

you may accidentally overwrite a file which is valuable. If you prefer that the C-shell not overwrite files in this way you can:

```
set noclobber
```

in your *.login* file. Then entering:

```
date > now
```

causes an error message if the file *now* already exists. You can enter:

```
date >! now
```

if you really want to overwrite the contents of *now*. The ">!" is a special syntax indicating that overwriting or "clobbering" the file is ok. (The space between the exclamation point (!) and the word "now" is critical here, as "!now" would be an invocation of the history mechanism, described below, and have a totally different effect.)

Using the C-Shell History List

The C-shell can maintain a history list into which it places the text of previous commands. It is possible to use a notation that reuses commands, or words from commands, in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the C-shell. Boldface indicates user input:

```
% cat bug.c
main()
{
    printf("hello);
}
% cc !$
cc bug.c
bug.c(4) :error 1: newline in constant
%  ed !$
ed bug.c
28
3s/);/"&/p
    printf("hello");
w
29
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
29
3s/lo/lo\\n/p
    printf("hello\n");
w
31
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 5124 + 614 + 1254 = 6692 = 0x1b50
bug: 5124 + 616 + 1252 = 6692 = 0x1b50
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill   7648 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill   7650 Dec 19 09:42 bug
% bug
hello
% pr bug.c | lpt
lpt: Command not found.
% ^lpt^lpr
pr bug.c | lpr
%
```

Figure 6-1 Sample History Session

In this example, we have a very simple C program that has a bug or two in the file *bug.c*, which we **cat** out on our terminal. We then try to run the C compiler on it, referring to the file again as "!$", meaning the last argument to the previous command. Here the exclamation mark (!) is the history mechanism invocation metacharacter, and the dollar sign ($) stands for the last argument, by analogy to the dollar sign in the editor which stands for the end-of-line.

The C-shell echoed the command, as it would have been typed without use of the history mechanism, and then executed the command. The compilation yielded error diagnostics, so we now edit the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as "!c", which repeats the last command that started with the letter "c".

If there were other commands beginning with the letter "c" executed recently, we could have said "!cc" or even "!cc:p" which prints the last command starting with "cc" without executing it, so that you can check to see whether you really want to execute a given command.

After this recompilation, we ran the resulting *a.out* file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra "-o bug" telling the compiler to place the resultant binary in the file *bug* rather than *a.out*. In general, the history mechanisms may be used anywhere in the formation of new commands, and other characters may be placed before and after the substituted commands.

We then ran the **size** command to see how large the binary program images we have created were, and then we ran an "ls -l" command with the same argument list, denoting the argument list:

```
!*
```

Finally, we ran the program *bug* to see that its output is indeed correct.

To make a listing of the program, we ran the **pr** command on the file *bug.c*. In order to print the listing at a lineprinter we piped the output to **lpr**, but misspelled it as "lpt". To correct this we used a C-shell substitute, placing the old text and new text between caret (^) characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with:

```
!!
```

and sent its output to the lineprinter.

There are other mechanisms available for repeating commands. The **history** command prints out a numbered list of previous commands. You can then refer to these commands by number. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in **csh**(C) the *XENIX Reference*.

Using Aliases

The C-shell has an alias mechanism that can be used to make transformations on commands immediately after they are input. This mechanism can be used to simplify the commands you enter, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using C-shell command files, but these take place in another instance of the C-shell and cannot directly affect the current C-shell's environment or involve commands such as **cd** which must be done in the current C-shell.

For example, suppose there is a new version of the mail program on the system called *newmail* that you wish to use instead of the standard mail program *mail*. If you place the C-shell command

```
alias mail newmail
```

in your *.cshrc* file, the C-shell will transform an input line of the form:

```
mail bill
```

into a call on *newmail*. Suppose you wish the command **ls** to always show sizes of files, that is, to always use the **-s** option. In this case, you can use the **alias** command to do:

```
alias ls ls -s
```

or even:

```
alias dir ls -s
```

creating a new command named **dir**. If we then enter:

```
dir ~bill
```

the C-shell translates this to:

```
ls -s /usr/bill
```

Note that the tilde (~) is a special C-shell symbol that represents the user's home directory.

Thus the **alias** command can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of

other commands. It is also possible to define aliases that contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism.

Thus the definition:

```
alias cd 'cd \!* ; ls '
```

specifies an **ls** command after each **cd** command. We enclosed the entire alias definition in single quotation marks (') to prevent most substitutions from occurring and to prevent the semicolon (;) from being recognized as a metacharacter. The exclamation mark (!) is escaped with a backslash (\) to prevent it from being interpreted when the alias command is entered. The "\!*" here substitutes the entire argument list to the prealiasing **cd** command; no error is given if there are no arguments. The semicolon separating commands is used here to indicate that one command is to be done and then the next. Similarly the following example defines a command that looks up its first argument in the password file.

```
alias whois 'grep \!^ /etc/passwd'
```

The C-shell currently reads the *.cshrc* file each time it starts up. If you place a large number of aliases there, C-shells will tend to start slowly. You should try to limit the number of aliases you have to a reasonable number (10 or 15 is reasonable). Too many aliases causes delays and makes the system seem sluggish when you execute commands from within an editor or other programs.

Redirecting Input and Output

In addition to the standard output, commands also have a diagnostic output that is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally useful to direct the diagnostic output along with the standard output. For instance, if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can enter:

command > & *file*

The "> &" here tells the C-shell to route both the diagnostic output and the standard output into *file*. Similarly you can give the command:

command |& lpr

to route both standard and diagnostic output through the pipe to the lineprinter. The form:

command >&! *file*

is used when "noclobber" is set and *file* already exists. Finally, use the form:

command >> *file*

to append output to the end of an existing file. If "noclobber" is set, then an error results if *file* does not exist, otherwise the C-shell creates *file*. The form:

command >>! *file*

lets you append to a file even if it does not exist and "noclobber" is set.

Creating Background and Foreground Jobs

When one or more commands are entered together as a pipeline or as a sequence of commands separated by semicolons, a single job is created by the C-shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line entered to the C-shell creates a job. Each of the following lines creates a job:

```
sort < data
ls -s | sort -n | head -5
mail harold
```

If the ampersand metacharacter (&) is entered at the end of the commands, then the job is started as a background job. This means that the C-shell does not wait for the job to finish, but instead, immediately prompts for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the C-shell. Thus:

```
du > usage &
```

runs the *du* program, which reports on the disk usage of your working directory, puts the output into the file *usage* and returns immediately with a prompt for the next command without waiting for *du* to finish. The *du* program continues executing in the background until it finishes, even though you can enter and execute more commands in the mean time. Background jobs are unaffected by any signals from the keyboard such as the **INTERRUPT** or **QUIT** signals.

The **kill** command terminates a background job immediately. Normally, this is done by specifying the process number of the job you want killed. Process numbers can be found with the **ps** command.

Using Built-In Commands

This section explains how to use some of the built-in C-shell commands.

The **alias** command described above is used to assign new aliases and to display existing aliases. If given no arguments, **alias** prints the list of current aliases. It may also be given one argument, such as to show the current alias for a given string of characters. For example:

```
alias ls
```

prints the current alias for the string "ls".

The **history** command displays the contents of the history list. The numbers given with the history events can be used to reference previous events that are difficult to reference contextually. There is also a C-shell variable named "prompt". By placing an exclamation point (!) in its value the C-shell will substitute the number of the current command in the history list. You can use this number to refer to a command in a history substitution. For example, you could enter:

```
set prompt=\! % ´
```

Note that the exclamation mark (!) had to be escaped here even within back quotes.

The **logout** command is used to terminate a login C-shell that has "ignoreeof" set.

The **rehash** command causes the C-shell to recompute a table of command locations. This is necessary if you add a command to a directory in the current C-shell's search path and want the C-shell to find it, since otherwise the hashing algorithm may tell the C-shell that the command wasn't in that directory when the hash table was computed.

The **repeat** command is used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could enter:

```
repeat 5 cat one >> five
```

The **setenv** command can be used to set variables in the environment. Thus:

```
setenv TERM adm3a
```

sets the value of the environment variable "TERM" to "adm3a". The program *env* exists to print out the environment. For example, its output might look like this:

```
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
```

The **source** command is used to force the current C-shell to read commands from a file. Thus:

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file that you wish to take effect before the next time you login.

The same holds true when using the **source** command with the *.login* file. The **time** command is used to cause a command to be timed no matter how much CPU time it takes. Thus:

```
time cp /etc/termcap /usr/bill/termcap
```

displays:

```
0.0u 0.4s 0:02 21%
```

Similarly:

```
time wc /etc/termcap /usr/bill/termcap
```

displays:

```
  2071   5849  92890 /etc/termcap
  2071   5849  92890 /usr/bill/termcap
  4142  11698 185780 total
1.3u 0.7s 0:04 47%
```

This indicates that the **cp** command used a negligible amount of user time (u) and about 4/10ths of a second of system time (s); the elapsed time was 2 seconds (0:02). The word count command **wc** used 1.3 seconds of user time and 0.7 seconds of system time in 4 seconds of elapsed time. The percentage "47%" indicates that over the period when it was active the **wc** command used an average of 47 percent of the available CPU cycles of the machine.

The **unalias** and **unset** commands are used to remove aliases and variable definitions from the C-shell.

Creating Command Scripts

It is possible to place commands in files and to cause C-shells to be invoked to read and execute commands from these files, which are called C-shell scripts. This section describes the C-shell features that are useful when creating C-shell scripts.

Using the argv Variable

A **csh** command script may be interpreted by saying:

```
csh script argument ...
```

where *script* is the name of the file containing a group of C-shell commands and *argument* is a sequence of command arguments. The C-shell places these arguments in the variable "argv" and then begins to read commands from *script*. These parameters are then available through the same mechanisms that are used to reference any other C-shell variables.

If you make the file *script* executable by doing:

```
chmod 755 script
```

or:

```
chmod +x script
```

and then place a C-shell comment at the beginning of the C-shell script (i.e., begin the file with a number sign (#)) then */bin/csh* will automatically be invoked to execute *script* when you enter:


```
script
```

If the file does not begin with a number sign (#) then the standard shell */bin/sh* will be used to execute it.

Substituting Shell Variables

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism known as variable substitution is performed on these words. Keyed by the dollar sign ($), this substitution replaces the names of variables by their values. Thus:

```
echo $argv
```

when placed in a command script would cause the current value of the variable "argv" to be echoed to the output of the C-shell script. It is an error for "argv" to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation:

```
$?name
```

expands to 1 if *name* is set or to 0 if *name* is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation:


```
$#name
```

expands to the number of elements in the variable "name". To illustrate, examine the following terminal session (input is in boldface):

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable that has several values. Thus:

```
$argv[1]
```

gives the first component of "argv" or in the example above "a". Similarly:

```
$argv[$#argv]
```

would give "c". Other notations useful in C-shell scripts are:

```
$n
```

where *n* is an integer. This is shorthand for:

```
$argv[ n ]
```

the *n*'th parameter and:

```
$*
```

which is a shorthand for:

```
$argv
```

The form:

```
$$
```

expands to the process number of the current C-shell. Since this process number is unique in the system, it is often used in the generation of unique temporary filenames.

One minor difference between "$*n*" and "$argv[*n*]" should be noted here. The form: "$argv[*n*]" will yield an error if *n* is not in the range 1-$#argv while "$n" will never yield an out-of-range subscript error. This is for compatibility with the way older shells handle parameters.

Another important point is that it is never an error to give a subrange of the form: "n-"; if there are less than "n" components of the given variable then no words are substituted. A range of the form: "m-n" likewise returns an empty vector without giving an error when "m" exceeds the number of elements of the given variable, provided the subscript "n" is in range.

Using Expressions

To construct useful C-shell scripts, the C-shell must be able to evaluate expressions based on the values of variables. In fact, all the arithmetic operations of the C language are available in the C-shell with the same precedence that they have in C. In particular, the operations "==" and "!=" compare strings and the operators "&&" and "| |" implement the logical AND and OR operations.

The C-shell also allows file inquiries of the form:

-? *filename*

where question mark (?) is replaced by a number of single characters. For example, the expression primitive:

-e *filename*

tells whether *filename* exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or if it has nonzero length.

It is possible to test whether a command terminates normally, by using a primitive of the form:

{ *command* }

which returns 1 if the command exits normally with exit status 0, or 0 if the command terminates abnormally or with exit status nonzero. If more detailed information about the execution status of a command is required, it can be executed and the "status" variable examined in the next command. Since "$status" is set by every command, its value is always changing.

For the full list of expression components, see **csh**(C) in the *XENIX Reference.*

Using the C-Shell: A Sample Script

A sample C-shell script follows that uses the expression mechanism of the C-shell and some of its control structures:

```
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i != *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```


This script uses the **foreach** command, which iteratively executes the group of commands between the **foreach** and the matching **end** statements for each value of the variable "i". I f you want to look more closely at what happens during execution of a **foreach** loop, you can use the debug command **break** to stop execution at any point and the debug command **continue** to resume execution. The value of the iteration variable (*i* in this case) will stay at whatever it was when the last **foreach** loop was completed.

The "noglob" variable is set to prevent filename expansion of the members of "argv". This is a good idea, in general, if the arguments to a C-shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a "$" variable expansion, but this is harder and less reliable.

The other control construct is a statement of the form:

```
if ( expression ) then
        command
        ...
endif
```

The placement of the keywords in this statement is not flexible due to the current implementation of the C-shell. The following two formats are not acceptable to the C-shell:

```
if ( expression ) # Won't work!
then
        command
        ...
endif
```

and:

```
if (expression) then command endif # Won't work
```

The C-shell does have another form of the if statement:

```
if ( expression ) command
```

which can be written:

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve "|", "&" or ";" and must not be another control command. The second form requires the final backslash (\) to immediately precede the end-of-line.

The more general **if** statements above also admit a sequence of **else-if** pairs followed by a single **else** and an **endif**, for example:

```
if ( expression ) then
        commands
else if ( expression ) then
        commands
...
else
        commands
endif
```

Another important mechanism used in C-shell scripts is the colon (:) modifier. We can use the modifier **:r** here to extract the root of a filename or **:e** to extract the extension. Thus if the variable "i" has the value */mnt/foo.bar* then

```
echo $i $i:r $i:e
```

produces:

```
/mnt/foo.bar /mnt/foo bar
```

This example shows how the **:r** modifier strips off the trailing ".bar" and the **:e** modifier leaves only the "bar". Other modifiers take off the last component of a pathname leaving the head **:h** or all but the last component of a pathname leaving the tail **:t**. These modifiers are fully described in the **csh**(C) page in the *XENIX Reference*. It is also possible to use the command substitution mechanism to perform modifications on strings to then reenter the C-shell environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the colon (:) modification mechanism. It is also important to note that the current implementation of the C-shell limits the number of colon modifiers on a "$" substitution to 1. Thus:

```
% echo $i $i:h:t
```

produces:

```
/a/b/c /a/b:t
```

and does not do what you might expect.

Finally, we note that the number sign character (#) lexically introduces a C-shell comment in C-shell scripts (but not from the terminal). All subsequent characters on the input line after a number sign are discarded by the C-shell. This character can be quoted using "´" or "\" to place it in an argument word.

Using Other Control Structures

The C-shell also has control structures **while** and **switch** similar to those of C. These take the forms:

```
while ( expression )
        commands
end
```

and:

```
switch ( word )

case str1:
        commands
        breaksw

...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see the manual section for **csh**(C). C programmers should note that we use **breaksw** to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in C-shell scripts is to use **break** rather than **breaksw** in switches.

Finally, the C-shell allows a **goto** statement, with labels looking like they do in C:

```
loop:
        commands
        goto loop
```

Supplying Input to Commands

Commands run from C-shell scripts receive by default the standard input of the C-shell which is running the script. It allows C-shell scripts to fully participate in pipelines, but mandates extra notation for commands that are to take inline data.

Thus we need a metanotation for supplying inline data to commands in C-shell scripts. For example, consider this script which runs the editor to delete leading blanks from the lines in each argument file:

```
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << ´EOF´
1,$s/^[ ]*//
w
q
´EOF´
end
```

The notation:

```
<< ´EOF´
```

means that the standard input for the **ed** command is to come from the text in the C-shell script file up to the next line consisting of exactly EOF. The fact that the EOF is enclosed in single quotation marks (´), i.e., it is quoted, causes the C-shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the "<<" which the C-shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form "1,$" in our editor script we needed to insure that this dollar sign was not variable substituted. We could also have insured this by preceding the dollar sign ($) with a backslash (\), i.e.:

```
1,\$s/^[ ]*//
```

Quoting the EOF terminator is a more reliable way of achieving the same thing.

Catching Interrupts

If our C-shell script creates temporary files, we may wish to catch interruptions of the C-shell script so that we can clean up these files. We can then do:

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the C-shell will do a "goto label" and we can remove the temporary files, then do an **exit** command (which is built in to the C-shell) to exit from the C-shell script. If we wish to exit with nonzero status we can write:

```
exit (1)
```

to exit with status 1.

Using Other Features

There are other features of the C-shell useful to writers of C-shell procedures. The **verbose** and **echo** options and the related **-v** and **-x** command line options can be used to help trace the actions of the C-shell. The **-n** option causes the C-shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that the C-shell will not execute C-shell scripts that do not begin with the number sign character (#), that is C-shell scripts that do not begin with a comment.

There is also another quotation mechanism using the double quotation mark ("), which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as the single quote (') does.

Starting a Loop at a Terminal

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if there were three shells in use on a particular system, */bin/sh*, */bin/nsh*, and */bin/csh*, you could count the number of persons using each shell by using the following commands:

```
grep -c csh$ /etc/passwd
grep -c nsh$ /etc/passwd
grep -c -v /sh$ /etc/passwd
```

Because these commands are very similar we can use **foreach** to simplify them:

```
$ foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
```

Note here that the C-shell prompts for input with "? " when reading the body of the loop. This occurs only when the **foreach** command is entered interactively.

Also useful with loops are variables that contain lists of filenames or other words. For example, examine the following terminal session:


```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
```

The **set** command here gave the variable "a" a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within back quotation marks (`) is converted by the C-shell to a list of words. You can also place the quoted string within double quotation marks (") to take each (nonempty) line as a component of the variable. This prevents the lines from being split into words at blanks and tabs. A modifier :x exists which can be used later to expand each component of the variable into another variable by splitting the original variable into separate words at embedded blanks and tabs.

Using Braces with Arguments

Another form of filename expansion involves the characters, "{" and "}". These characters specify that the contained strings, separated by commas (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus:

```
A{str1,str2,...strn}B
```

expands to:

```
AstrlB Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e., nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be:

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories *hdrs*, *retrofit* and *csh* in your home directory. This mechanism is most useful when the common prefix is longer than in this example:

```
chown root /usr/demo/{file1,file2,...}
```

Substituting Commands

A command enclosed in accent symbols (`) is replaced, just before filenames are expanded, by the output from that command. Thus, it is possible to do:

```
set pwd=`pwd`
```

to save the current directory in the variable "pwd" or to do:

```
vi `grep -l TRACE *.c`
```

to run the editor **vi** supplying as arguments those files whose names end in which have the string "TRACE" in them. Command expansion also occurs in input redirected with "<<" and within quotation marks ("). Refer to **csh**(C) in the *XENIX Reference* for more information.

Special Characters

The following table lists the **csh** and XENIX special characters. A number of these characters also have special meaning in expressions. See the **csh** manual section for a complete list.

Syntactic metacharacters

;	Separates commands to be executed sequentially
|	Separates commands in a pipeline
( )	Brackets expressions and variable values
&	Follows commands to be executed without waiting for completion

Filename metacharacters

/	Separates components of a file's pathname
.	Separates root parts of a filename from extensions
?	Expansion character matching any single character
*	Expansion character matching any sequence of characters
[ ]	Expansion sequence matching any single character from a set of characters
~	Used at the beginning of a filename to indicate home directories
{ }	Used to specify groups of arguments with common parts

Quotation metacharacters

\	Prevents meta-meaning of following single character
'	Prevents meta-meaning of a group of characters
"	Like ', but allows variable and command expansion

Input/output metacharacters

< Indicates redirected input

> Indicates redirected output

Expansion/Substitution Metacharacters

$ Indicates variable substitution

! Indicates history substitution

: Precedes substitution modifiers

^ Used in special forms of history substitution

` Indicates command substitution

Other Metacharacters

Begins scratch filenames; indicates C-shell comments

- Prefixes option (flag) arguments to commands

Chapter 7

The Korn Shell

Introduction

The KornShell, **ksh**(C), is an interactive command-language interpreter and programming language that reads and executes commands from either the terminal or a file. The **ksh** combines the best features of the two common XENIX System shells, the standard Bourne shell, **sh**(C), and the C shell, **csh**(C). The **ksh** provides both compatibility with **sh** and the command history and substitution features of **csh**. In addition, **ksh** includes command-line editing, and enhanced command-history functionality.

Starting ksh

If you are currently running **sh** or **csh**, you can start **ksh** by entering **ksh** at the command line. When you run **ksh** as a program from your original login shell, you cannot automatically access the command-line editing feature.

To use **ksh** as your default login shell, ask the system administrator to change your login-shell specifier in the */etc/passwd* file to **ksh**. When the system administrator specifies **ksh** as the login shell when creating a new user, the **sysadmsh**(ADM) utility creates two files in the user's home directory: *.profile* and *.kshrc*.

When you log in using **ksh** as your login shell, the shell reads commands from the system profile file, */etc/profile*, and then from *.profile* in the current directory or $HOME/*.profile*, if either file exists. Next, the shell reads commands from the **ksh** environment file, $HOME/*.kshrc*, if it exists. If there is no *.sh_history* file (the history file where **ksh** stores commands that you enter at the keyboard) in the your home directory, **ksh** creates one.

For more information on these files and how to modify your **ksh** environment, see the section later in this chapter, "Customizing the ksh Environment."

Using the ksh Built-in Editors

Using **csh** or **sh**, the only way to fix errors on the command line is to backspace or retype the entire line. With **ksh**, you can edit the command line using the familiar commands that you use to edit files. The **ksh** provides both **vi**-like and **emacs**-like built-in editor interfaces for editing the command line.

At login time, **ksh** reads the *.kshrc* environment file and turns on the **vi**-like editor. You can turn off the **ksh** editor functionality completely or turn off **vi** and turn on **emacs** for the current session or for each login session.

To turn off **vi** for the current login session only, enter the following at the command line:

```
set +o vi
```

To turn on **emacs** for the current login session, enter the following at the command line:

```
set -o emacs
```

To turn on or off either the **vi** or **emacs** editors automatically when you log in, add the appropriate command to the *.profile* or environment file (*.kshrc* by default).

You can also use the EDITOR and VISUAL environment variables to set the editor to any pathname that ends in **vi**. For example, to turn on the **vi** editor automatically when you log in, add the following line to your *.kshrc* file:

```
EDITOR=/usr/bin/vi
```

Note

The VISUAL variable overrides the EDITOR variable.

This chapter includes information on the built-in **vi**-like editor. For information about using the **emacs**-like editor, see your **emacs** documentation.

Using the vi Built-In Editor Modes

Like the **vi** text editor, **ksh**'s built-in **vi** editor has two modes: *input mode* and *control mode*. In input mode, **ksh** inserts the characters that you type at the keyboard in an editing buffer. In control mode, **ksh** interprets the characters that you enter at the keyboard as editing commands.

When you log in using **ksh** as your login shell, you are in input mode automatically. (This differs from the **vi** text editor; you are initially in control mode and you must press **a** or **i** to begin entering text.) To enter control mode from input mode, press ⟨Esc⟩. If you press ⟨Esc⟩ while in control mode, the terminal beeps.

Editing in Input Mode

While entering commands in input mode, you can edit the command line using editing commands from the following table:

Input Mode Editing Commands

Command	Description
⟨Ctrl⟩**h** or ⟨Bksp⟩	moves back one character
⟨Return⟩ or ⟨Ctrl⟩**m**	executes the current line
⟨Ctrl⟩**v**	escapes the character that follows (for entering control characters)

Editing in Control Mode

At any time before you press ⟨Return⟩ to execute the command, you can press ⟨Esc⟩ to enter control mode. In control mode, you can move around the command line as if you were in **vi**, editing a file.

The following table shows the **vi** commands for moving the cursor on the command line in control mode:

Moving the Cursor

Key	Description
h	moves left one character
l	moves right one character
b	moves left one word
B	moves left one word, skipping punctuation
w	moves right one word
W	moves right one word, skipping punctuation
e	moves to the last character of the next word
E	moves to the last character of the next word, skipping punctuation
0	moves to the beginning of the current line
$	moves to the end of the current line
^	moves to the first character on the current line that is not a ⟨Space⟩ or ⟨Tab⟩
f*x*	moves right to the next occurrence of *x*
F*x*	moves left to the preceding occurrence of *x*
t*x*	moves right to the character before the next occurrence of *x*
T*x*	moves left to the character following the preceding occurrence of *x*
;	repeats the last character search **f**, **F**, **t**, or **T**.
,	reverses the last character search **f**, **F**, **t**, or **T**.

The following table gives the commands for entering input mode from control mode, and for changing and deleting text:

Adding, Changing, and Deleting Text

Key	Description
a	enters input mode after the character under the cursor
A	enters input mode after the last character on the line
i	enters input mode before the character under the cursor
I	enters input mode before the first character on the line
_	appends the last word of the previous **ksh** command to the current line and then enters input mode.
r*z*	replaces the character under the cursor with *z*
R*text*	replaces characters with *text* beginning at the cursor
c*motion*	changes the characters from cursor position, using the **vi** *motion* command For example:
cw	changes word below cursor
cl	changes character below cursor and then adds text
c$	changes from the current character to the end of line
cc	deletes the entire line and returns to input mode (same as c$)
x	deletes the character under the cursor
X	deletes the character to the left of the cursor
dw	Deletes the word under cursor
d*motion*	deletes characters, starting at the cursor, up to and including the other end of *motion*
D	deletes from the cursor to the end of line
d$	same as **D**

(*Continued on the next page.*)

Adding, Changing, and Deleting Text (*Continued*)

Key	Description
dd	deletes the entire line
y*motion*	yanks the current character using **vi** motion command
Y	yanks from cursor to end of line
y$	same as **Y**
yy	yanks the entire line into the buffer
p	puts previously yanked (or deleted) words to the right of the cursor
P	puts previously yanked (or deleted) words to the left of the cursor

The following table shows the control mode commands for executing and redrawing the current line, repeating commands, and undoing modifications on the command line:

Miscellaneous Control Mode Commands

Command	Description
⟨Return⟩ or ⟨Ctrl⟩**m**	executes the current line
⟨Ctrl⟩**l**	redraws the current line
~	changes the case of the character under the cursor
.	repeats the most recent **vi** command
u	undoes the previous **vi** command
U	undoes all modifications on the current line

Accessing Commands in the History File

Using the **vi** built-in editor, you can access previously entered commands that are stored in your **ksh** history file (*.sh_history* by default). Once you retrieve a command, you can modify it and execute it again.

Displaying Commands in the History File

To display the list of the commands that are stored in the history file, enter **history**. The **history** command is a predefined alias that uses the **ksh** built-in command, **fc** (fix command), to access the history file. For more information about **fc**, see **ksh**(C).

The **history** alias displays the last 16 (or fewer, if there are fewer than 16 commands in the file) commands in the history file. You can specify how many and which commands that you want **history** to display. Note that the commands must be accessible in the history file for **history** to display them. The following list gives examples of how to use the **history** alias:

history -4	displays the previous four commands only.
history 20	displays all commands from the history file, starting with 20.
history 12 24	displays only commands 12 through 24.

Reexecuting Previous Commands

The **ksh** also includes **r**, another predefined alias that uses the **fc** built-in command. The **r** alias allows you to reexecute commands from the history file. This alias functions similarly to the ! command in **csh**. The following list shows some common uses of the **r** alias:

Alias	Description
r	reexecutes the last command entered
r *command*	reexecutes the last *command* entered
r *x*	reexecutes the last command beginning with *x*
r #	reexecutes command number #

Note that **r** simply reexecutes commands from the history list; **r** does not allow you to modify commands before you execute them.

Editing Previous Commands

You can use **vi** commands to search for and retrieve commands from the history file. Once you locate a command, you can edit and reexecute it using the **vi** commands described in the section "Editing in Control Mode" earlier in this chapter.

To move through the history file, first press ⟨Esc⟩ to enter control mode. Then, use the **vi** commands in the following table to move up and down in the history file:

Moving in the History File

Command	Description
k	moves up (previous) one command in the history file
j	moves down (next) one command in history file
/string	searches left and up (back) through the history file for the next command containing *string*
?*string*	searches right and down (forward) through the history file for the next command containing *string*
G	goes back to the oldest accessible command in the history file
n	repeats the last / or ? search command
N	repeats the last / or ? command, searching backward

Customizing the ksh Environment

The **ksh** uses two files located in your home directory, *.profile* and *.kshrc*, to set up your environment. Use the *.profile* file to set variables and options for your login shell and other programs that you run from the **ksh**. The *.kshrc* file is the **ksh**-specific environment file; use it to define aliases and set **ksh** command-line options.

Modifying the .profile File

Whenever you log in using **ksh** as your login shell, **ksh** executes the *.profile* file, automatically executing commands and setting exported environment variables. Commands and environment variables in *.profile* must be in the same format as they are when you enter them from the keyboard.
To assign values to **ksh** environment variables, use this format:

```
EDITOR=/bin/vi
export EDITOR
```

The following table shows some of the more common environment variables:

Environment Variables

Variable	Description
COLUMNS	specifies the number of columns that **ksh** uses to display the command line
EDITOR	sets either the **vi**-like or **emacs**-like editor to use when editing the command line
ENV	sets the environment file (*.kshrc* by default)
HISTFILE	sets an alternate history file (*.sh_history* by default)
HISTSIZE	sets the maximum number of commands that are stored in the history file (128 by default)
HOME	specifies the default argument used by the **cd**(C) command
MAILCHECK	specifies the interval in seconds that **ksh** checks for new mail (600 seconds by default)
PATH	specifies the pathnames that **ksh** searches when executing commands
PS1	specifies the primary prompt to display when the **interactive** option is on ($ by default); **ksh** replaces an exclamation point (!) with the command number (to print a ! in the prompt, enter !!)
TERM	specifies the type of terminal that you are using
VISUAL	sets the editor to use when editing command lines (overrides the value of EDITOR)

For example, to set up your prompt to include the machine name and command number, add the following following lines to your *.profile*:

```
PS1='! fscott'
export PS1
```

For a complete list of environment variables used by **ksh**, see **ksh**(C).

Executing a File on Logout

You can use the **trap 0** command in your *.profile* file to instruct **ksh** to execute a file, for example, *.logout*, when you exit the shell. To do this, enter the following in *.profile*:

```
trap $HOME/.logout 0
```

The *.logout* file must be executable.

Modifying the .kshrc File

The *kshrc* file contains definitions for aliases and functions and default option settings for **ksh**. You should specify any commands and definitions that only **ksh** recognizes in this file rather than in *.profile*.

Defining Aliases

An alias is an abbreviated name for a command. Use the following format to define an alias:

alias *shortname='commandname'*

where *shortname* is the abbreviated name for *commandname*. For example, to define **ls** as an alias for **lc -F**, use the following command:

```
alias ls='lc -F'
```

You can define aliases for the current **ksh** session by entering **alias** at the command line. To specify that an alias definition remain in effect across login sessions, add the alias command to your *.kshrc* (or other **ksh** environment file).

You can display the complete list of your aliases by entering **alias** at the prompt. To display a particular alias definition, enter **alias** followed by the alias name. For example, if you enter **alias ls**, **ksh** displays:

```
ls=lc -F
```

To unset a particular alias (in either the *.kshrc* file or at the command line), enter **unalias** followed by the alias name.

When you run scripts that do not invoke another **ksh**, regular aliases do not remain defined. However, you can *export* these alias definitions by defining them in your environment file using this format:

```
alias -x ls='lc -F'
```

The **ksh** automatically predefines several aliases. These aliases are compiled into the shell, but you can unset or redefine them. (We do not recommend redefining preset aliases.) The two most common preset aliases are **history** (for displaying the contents of the *.sh_history* file) and **r** (for reexecuting previously entered commands). For more information about preset aliases, see the section "Accessing Commands in the History File" in this chapter and **ksh**(C).

Setting ksh Options

You can set **ksh** command-line options in your environment file. Use the following format:

```
set -o option
```

To unset an option, use a plus (+) character in place of minus (-). The following table lists some useful **ksh** options:

set Options

Option	Description
allexport	exports all subsequent variables automatically (same as **-a**)
bgnice	runs all background jobs at a lower priority (set by default)
emacs	uses the **emacs**-like built-in editor for command-line editing
ignoreeof	prevents **ksh** from exiting on end-of-file, ⟨Ctrl⟩**d**; (when **ignoreeof** is set, you must enter **exit** to terminate the shell)
verbose	prints shell input lines as they are read
vi	uses the **vi**-like built-in editor for command-line editing

For a complete list of **set** options and descriptions, see **ksh**(C).

Modifying the ksh History File

The **ksh** stores the commands that you enter at the keyboard in the *.sh_history* file in your home directory by default. You can specify a different history file using the HISTFILE environment variable in your *.profile* file.

For example, to use the file *.history* instead of *.sh_history*, add the following lines to your *.profile*:

```
HISTFILE=~/.history
export HISTFILE
```

You can specify the maximum number of previously entered commands that you can retrieve from the history file with the HISTSIZE environment variable. If HISTSIZE is not set, **ksh** stores 128 commands by default. There is no limit to the number of commands that the **ksh** can store.

Note

If HISTSIZE is very large, **ksh** may be very slow at startup time.

The **ksh** does not delete the history file when you exit; the shell appends and stores commands across login sessions. When you log in, **ksh** deletes any commands in your history file that are older than the last number of commands specified by HISTSIZE.

Manipulating Commands Wider Than the Screen

The **ksh** allows you to enter commands of up to 256 characters from the terminal. You can define the maximum width of the command-line display (80 columns by default) using the COLUMNS variable. (See the section "Customizing the ksh Environment" in this chapter for more information about setting environment variables.)

If you edit a command that is wider than the command-line display minus two columns, **ksh** automatically scrolls the command line horizontally to the left or right of your screen. In the last column on the right side of the screen, **ksh** displays one of the following characters to show that the line is scrolling:

- < scrolls to the right (text to the left is not displayed)
- > scrolls to the left (text to the right is not displayed)
- * text both to the right and to the left is not displayed

For example, if COLUMNS is not set (the width of the command-line display is the default of 80 columns) and your command line is greater than 78 characters wide, **ksh** scrolls the command line left to display the end of the line. To the right of the command line, **ksh** displays the > character.

Horizontal scrolling does not work when you initially enter a command.

Using Expanded cd Capabilities

The **ksh** includes expanded functionality for the **cd**(C) command. You can instruct **ksh** to search through a specified list of directories when you enter pathnames that do not begin with the slash (/) character. To do this, set the CDPATH variable in your *.profile* file. (See the section "Customizing the ksh Environment" for more information about setting environment variables.)

The **ksh** provides an option to **cd** that allows you to return quickly to your previous working directory. For example, if you are in the */usr/spool/mail* directory and you enter:

cd /usr/bin

you can return to your previous directory, */usr/spool/mail*, by entering **cd -**. From this directory, you can enter **cd -** again to return to the */usr/spool/mail* directory.

The **ksh** provides a means for changing to a directory with a pathname that is slightly different from your current working directory. To do this, use the following format:

cd *old new*

where *old* is the part of the pathname that you want to change and *new* is what you want to change it to. For example, if you are in */usr/spool/mail* and you want to change to */usr/bin/mail*, enter:

cd spool bin

Chapter 8

The Visual Shell

What is the Visual Shell?

The visual shell, **vsh**, is a menu-driven XENIX shell. This chapter describes the use and behavior of the vsh. This chapter assumes that the reader is familiar with some general XENIX concepts, specifically the structure of XENIX filesystems and the nature of a XENIX "command". No familiarity with any other shell, however, is assumed. If you are a first-time user of the visual shell, please completely read the narrative sections of this chapter.

A "shell" is a program which passes a command to an operating system, and displays the result of running the command. The XENIX shells can also create "pipelines" for passing the output of one command to another command or "redirect" the output into a file.

The other XENIX shells available are the Bourne shell, the C-shell, and the Korn shell. These shells are called "command-line oriented" shells. This means that the user enters commands one line at a time. They are full computer languages which require study and some programming knowledge to use effectively. These command-line shells are powerful and efficient.

The **vsh** is a "menu-oriented" shell. In a menu-oriented shell, the user is given the available commands, or some of the available commands. The user can run the command, by selecting from the menu.

The visual shell is a good shell for users who may not want to master a programming language right away just to use XENIX or a specific XENIX application. All visual shell users should additionally become familiar with some command-line shell usage.

Users familiar with command-line shells are in for a pleasant surprise if they try the visual shell. Experienced users will appreciate the efficiency and versatility of the visual shell. The distinction is very much akin to the difference between a line-oriented text editor and a full-screen editor.

A menu shell can be used effectively with very little study. On the other hand, a menu shell can also restrict the user from using the operating system in creative, possibly more efficient ways. The Microsoft visual shell strikes a balance in this regard. The visual shell is designed to do all of the things that the command-line shells can do.

Getting Started with the Visual Shell

This section describes how to enter, obtain help about, and leave the visual shell. This section also describes what you see on the screen while running the visual shell and how the menus work.

Note the following convention for specifying keystrokes. ⟨Ctrl⟩ refers to the Ctrl key. ⟨Ctrl⟩c means pressing the Ctrl and "c" keys at the same time. Note the irrelevance of case in entering Menu Selection characters. For instance, press either **Q** or **q** to run the "Quit" command from the main menu.

Entering the Visual Shell

Log in to XENIX. If you are not sure how to log in, consult the *XENIX System Administrator's Guide* or have someone knowledgeable about XENIX help you. When you have a shell prompt (typically "$" or "%"), the operating system is waiting for a command. Enter the command:

vsh

and press ⟨Return⟩.

Getting Help

If at anytime you are not sure what to do, either run the "'Help'" Menu Selection or press the question mark (?), which is the "help key." Refer to the reference section of this chapter for information about the Help command.

Leaving the Visual Shell

To exit the visual shell select the **Quit** command from the main menu. The simplest way to do this is to simply press **q** or **Q**. In response to the prompt "Type Y to confirm", enter **y** or **Y**. If you don't want to exit the visual shell yet (perhaps you pressed "q" by mistake), enter any other character but "y" or "Y". If you have invoked the visual shell from another shell, as described above, you will need to log out from XENIX by entering ⟨Ctrl⟩d or **logout** and pressing ⟨Return⟩. If the visual shell is your default shell, you will automatically be logged out.

The Visual Shell Screen

Status Line

The bottom line on the screen is called the "status line". The status line displays the name of the current working directory, notifies you if you have mail, and gives the date, time and the name of the operating system.

Message Line

The line above the "status line" is called the "message line". The message line displays special output from XENIX commands, such as error reports.

Main Menu

The next section of the screen above the message line is the "Main menu". The Main menu displays a selection of useful XENIX commands.

The currently selected menu command is highlighted on the screen. To select any command, press the ⟨Space⟩. The next highlighted command is selected. The ⟨Bksp⟩ key will move to the previous command. Move through the menu until you have found the command you want. To run the currently selected command, press ⟨Return⟩.

You can also enter the first letter of a command to select that command. If you enter the first letter of the command, you do not need to press ⟨Return⟩.

If you enter a letter which does not correspond to a menu selection, the message:

```
Not a valid option
```

is displayed. Try another option.

Command Option Menu

When you have selected a command, the main menu is replaced with a command option menu. The command option menu gives the options available with the specific command. You must fill in the options with appropriate responses.

If you wish to return to the main menu without running the command, press ⟨Ctrl⟩c (cancel). If you want to run the command with the selected options press ⟨Return⟩.

The following keystrokes allow editing of option responses.

⟨Ctrl⟩i ⟨Ctrl⟩a, or ⟨Tab⟩,	Move to next field in options menu.
⟨Ctrl⟩y or ⟨Del⟩	Delete character under cursor.
⟨Ctrl⟩n	Move cursor to character to right of current position in current option field.
⟨Ctrl⟩b	Move cursor to character to left of current position in current option field.
⟨Ctrl⟩p	Move cursor to word in current field to right of the current word.
⟨Ctrl⟩o	Move cursor to word in current field to left of the current word.

Program Output

While running a command, commands given and output (unless redirected) are displayed above the menu and below the view window. The output *scrolls up:* moves from bottom to top. Lines scrolling off the top of the output window disappear.

Visual shell command lines are listed with each argument preceded by the number in the argument list enclosed in parentheses. The command is named in the output window by the menu command. Hence, if you run the command **/bin/ls** with the argument **-R**, the output window will display the command line as follows:

```
Run (1) /bin/ls (2) -R (3)
```

To change the command line format to reflect the actual XENIX command line generated by the visual shell, use the Options Output menu command.

View Window

A menu of currently accessible files and directories can be displayed at the top of the screen in alphabetical order, left to right, top to bottom. Note that this display is the same as that obtained using the view command. This will be referred to as the "view window" in this chapter. If the directory list is larger than the current window size, you may scroll through using the key commands given below. To reset the window size, use the "Window" Main menu command.

The currently selected item is highlighted in the view window. Use the arrow keys and other key commands given at the end of this section to move the highlight around the window.

If a directory is being listed, subdirectories are shown enclosed in square brackets. To view a subdirectory, press = while the directory is highlighted. To return to the previous directory after viewing a subdirectory, press -. The parent directory of the current directory is shown as "[..]." The current directory is shown as "[.]." Executable files are preceded by an asterisk. The last modification date of the currently selected item is given at the right margin of the last line of the window. The name of the item in view in the current window is given in the upper right-hand corner of the window.

The view window may also display contents of files. Highlight a file, and press =. You may scroll through the file using the key commands given below. While viewing a file, the highlighted area covers one line.

If you press "=" while an executable file is highlighted, that file will be run.

If the visual shell requires a file or directory name, the currently selected View Window item can be automatically entered in the relevant option field by pressing any directional movement key following selection of the command. This method saves keystrokes and reduces the chance of making a mistake while entering a command. On the other hand, if you wish to enter a file or directory in an option field, enter in the name after selecting the command.

Use these keystrokes to select files from the view window:

Table 8.1
Window Motion Keys

⟨Ctrl⟩q	Move to start (first item alphabetically) of view window.
⟨Ctrl⟩z	Move to end (last item alphabetically) of view window.
⟨Ctrl⟩r ⟨Ctrl⟩e	Scroll view window up.
⟨Ctrl⟩r ⟨Ctrl⟩s	Scroll view window down.
=	View indicated item, either file or directory. If no view window is present, the current working directory is displayed.
-	Return window display to parent directory of currently listed directory. If viewing a file, exit from viewing that file. Last view window is returned to.

Table 8.2
Directional Movement Keys

ARROW UP or ⟨Ctrl⟩e	Move highlight up in view window.
ARROW DOWN or ⟨Ctrl⟩x	Move highlight down in view window.
ARROW LEFT or ⟨Ctrl⟩s	Move highlight left in view window.
ARROW RIGHT or ⟨Ctrl⟩d	Move highlight right in view window.

Movement beyond the left or right margin will proceed to the next item on the previous or next line unless at the edge of the view window. Movement beyond the top or bottom edge of the current window will scroll the view window up or down if there are more items in that direction in the view window.

Note that there are two ways to move the highlight around. Either use the keypad arrow keys or the cluster of four keys on the far left of the keyboard "e", "x", "s", and "d" shifted with ⟨Ctrl⟩.

While viewing a file, the directional movement keys for up and left move the highlight up, and the keys for down and right move the highlighted line down.

Visual Shell Reference

The following is a reference of useful visual shell commands.

Visual Shell Default Menu

This section describes the default visual shell menu commands and options. The menu options are displayed at the bottom of the screen above the status line.

To invoke a command, move the highlight forward through the main menu using the space bar or the tab key, or backwards using the backspace key. Or simply press the first letter of the command.

Most commands require entering options. Move the cursor to the field using the ⟨Space⟩, ⟨Tab⟩ key or ⟨Bksp⟩ key, and enter your response. To edit the options, refer to the key commands listed above in the section in this chapter labeled "Command Option Menu". To select an item from a View Window listing for insertion in a field, refer to the section in this chapter labeled "View Window".

Note that some options have "switches" with predefined (default) selections. The currently selected switch setting is highlighted. The default is the parenthesized setting. For instance, in the switch:

```
Recursive: (yes) no
```

the default is "recursive." To change a switch, select the field and press the ⟨Space⟩ or ⟨Bksp⟩.

Copy

The Copy command can copy files and directories. To copy a file, select "File" from the options, to copy a directory, select "Directory". A sub-menu then appears. Enter the file or directory you wish copied in the *from:* field. Enter the file or directory you wish copied to the *to:* field. Note that if the item in the *to:* field already exists, it is overwritten, so be careful.

The Copy Directory sub-menu has a switch "recursive". If this switch is set to "yes," all sub-directories and their contents below the specified directory will be copied.

Delete

The Delete command can remove files and directories. In the *DELETE name:* field, enter the name of the file or directory you want to remove. Note that once the file or directory is deleted, the contents are permanently removed unless you have another copy, so be careful.

Edit

The Edit command invokes the full-screen editor **vi**. The current directory is displayed in the output window. Enter in the option field *EDIT filename:* the name of the file you wish to edit using **vi**.

To learn **vi**, refer to "vi: a Text Editor" in the XENIX *XENIX User's Guide*, and the **vi**(C) manual page in the *XENIX Reference*. A **vi** reference card is also available.

Help

The Help command (also available by pressing **?** at any time), can give online help regarding many aspects of visual shell use. The view window displays the help file. Use the menu to select the topic you need help with. For instance, move the highlight to "Keyboard" using the ⟨Space⟩ and press ⟨Return⟩ to view the help file starting at the "Keyboard" section. The "Next" and "Previous" fields in the menu will scroll through the the help file, from the present location, one screen at a time. Your work will remain undisturbed. To return from Help, press ⟨Ctrl⟩c or select the "Resume" menu option.

Mail

The Mail command enters the XENIX mail system. There are two options: "Send" and "Read" For more information about mail, refer to the section of the XENIX *Users Guide* titled "mail", or refer to the **mail**(C) manual page.

Name

The Name command renames an existing file or directory. There are two fields, *From:* and *To:*. Enter the name of the file or directory you want to rename in *From:* and the new name in *To:*.

Options

The Options Main Menu Selection provides four sub-menus. These sub-menus run commands which are used infrequently, or which have irrevocable results.

Directory Option

The Directory command has two sub-menus, Make and Usage.

Make Directory Option:

This command creates a new directory named what you enter in the *name:* field.

Usage Directory Option:

Counts the number of disk blocks in the directories specified in the *name:* field. The format is the same as the XENIX command **du.** Refer to the manual page **du**(C).

FileSystem Option

FileSystem has five sub-menus: Create, FilesCheck, SpaceFree, Mount and Unmount.

Create FileSystem Option:

Create FileSystem makes a XENIX filesystem. The Create command performs radical system maintenance and may have irrevocable effects. Care is advised when using Create FileSystem.

The functionality is the same as **mkfs**(ADM). Consult the **mkfs**(ADM) manual page before running Create FileSystem. Create FileSystem prompts you for device, block size, gap number and block number. Refer to the "Managing Filesystems", chapter in the XENIX *System Administrator's Guide*, for information on creating file systems.

FilesCheck FileSystem Option:

FilesCheck checks the consistency of a XENIX filesystem and attempts repair if damage is detected. The FilesCheck command performs radical system maintenance and may have irrevocable effects. Care is advised when using FilesCheck.

The functionality is the same as **fsck**(ADM). Consult the **fsck**(ADM) manual page before running FilesCheck. FilesCheck prompts you for the device to check.

Output Option:

The Output Option command has one switch, *commands like: VShell XENIX"*. The default is VShell. IF VShell is set, the **vsh** form of commands given appear in the upward scrolling output window. If XENIX is specified, the XENIX command line which **vsh** generated is shown instead.

Permissions Option

The Permissions Option command allows changing the access permissions on files and directories. The functionality is the same as the **chmod**(C) command. Consult the **chmod** manual page if you do not understand the concept of XENIX permissions.

In the *name:* field enter the name of the file or directory you wish to alter the permissions on. You may only alter the permissions on files and directories you own. There are four switches, *who:*, *read:*, *write:*, and *execute:*.

The *who:* switch has four settings, *All*, *Me*, *Group* and *Others*. *All* is the default. *All* refers to yourself, those with the same group id as yourself and others. *Me* refers to yourself. *Group* refers to all others with your group id. *Others* refers to those outside your group.

The *read:*, *write:*, and *execute:* switches have two settings, "yes" and "no". The default is "yes" for *Me*, and "no" for *Group* and *Others*. This grants the given type of permission to those specified in the *who:* switch. No takes away the given type of permission from those specified in the *who:* switch.

Print

The Print command puts a file or files in the queue for your printer. In the *filename:* option field, enter the file or files you want to print.

Quit

The Quit command exits the visual shell. The only option is *Enter Y to confirm:*. Enter **Y** or **y** if you really want to quit. Any other key cancels the quit.

Run

The Run command executes a program or shell script. The *name:* option takes the name of an executable file. In the *parameters:* option field enter flags to pass to the executable file. The *output:* option can specify a file to redirect output to, or another program to send the output to. Enter | (a vertical bar) in the output field to use the pipe menu.

It is also possible to run an executable file by highlighting the name of the file in the View Window and pressing =.

View

The View command allows you to inspect without altering the contents of files and directories. View is also available at any time for an item highlighted in the View Window by pressing =. See the section above labeled "View Window" for the details of using View.

To alter the height and characteristics of the View Window, use the "Window" menu option. See the section below labeled "Window."

If you have invoked View from the menu, enter the name of the file or directory you wish to view in the *VIEW name:* field, or select from a directory view window.

To return from any View action to the previously displayed View Window, press the minus key (-).

If you View a non-executable binary file, non-ascii characters are displayed as the character '@'.

Window

The Window command alters the height and redraw characteristics of the visual shell View Window.

The

```
WINDOW redraw: Yes (No)
```

switch turns redraw of the view window on or off after running a command.

The *height in lines:* field changes the number of lines displayed in the view window. The minimum window height is 1 line. The default window height is 5 lines. The maximum window height is 15 lines.

Pipes

XENIX allows output from one program to be passed to another program or to be put in a file. This is called "piping" or "pipelining." If the output is placed in a file it is said to be "redirected." Piping is supported in the visual shell through the pipe menu.

The Pipe menu is invoked by entering a vertical bar "|" character in any option field named *output:*. For instance, the Run main menu and the Pipe menu itself have an *output:* field. The available Pipe menu commands are Count, Get, Head, More, Run, Sort and Tail. Each Pipe menu sub-command also has an *output:* field, which allows construction of pipelines of arbitrary length.

Count

Count counts words, lines and characters in the input pipe. The default is all of the above. There is a switch for each type of item to count. The Count Pipe Menu option corresponds to the XENIX command **wc**. Consult the manual page **wc**(C) for an explanation.

Get

Get looks for patterns in the input pipe. The pattern is specified in the *GET lines containing* field. The pattern may be verbatim, or you may specify a "regular expression" to look for. Regular expressions may contain 'wildcard' characters which represent sets of strings. Consult the manual page **grep**(C), for the available wildcard characters.

The first Get switch is *Unmatched Yes (No).* If you specify No (the default), all lines containing the given pattern will be output. If you specify Yes, all lines not containing the given pattern are output.

The second Get switch is *ignore case:* which suppresses the case while looking for the regular expression. The default is off.

The third Get switch is *line numbers:*, which reports the line in the input stream which the regular expression was matched on. The default is on.

Head

Head prints a specified number of lines of the input stream starting from the first line. The *lines:* field may be set to specify the number of lines at the head of the input stream to print. The default is 5 lines.

The Head Pipe Menu option corresponds to the XENIX command head. Consult the manual page **head**(C) for an explanation.

More

More allows viewing an input stream one screen at a time. The More Pipe Menu option invokes the XENIX command more. Consult the manual page **more**(C) for an explanation.

Run

The Run Pipe Menu option allows the specification of any command not in the Pipe menu. The functionality is the same as the visual shell Main Menu Option "Run".

Sort

The XENIX sort utility can be invoked through the Sort Pipe menu option. The input stream is sorted.

The first Sort switch is *order:* < >. Select "<", the default, to sort in ascending order. Select ">" to sort in descending order.

The second Sort switch suppresses the case of characters in the sort. The default is off.

The third Sort switch sorts the input stream assuming an initial numeric field is in the input stream. If this switch is off, initial numbers are sorted in ascii order, which means that a line beginning with "10" will be output before the line beginning with "2." The default is off.

The fourth Sort switch sorts the input stream in alphabetical order, rather than ascii order.

The Sort Pipe Menu option corresponds to the XENIX command sort. Consult the manual page **sort**(C) for an explanation.

Tail

Tail prints a specified number of lines of the input stream up to the end of the stream. The *lines:* field may be set to specify the number of lines to print. The default is 15 lines.

The Tail Pipe Menu option corresponds to the XENIX command tail. Consult the manual page **tail**(C) for an explanation.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z


AZ01203P000
53804